Святослав Куликов

РЕЛЯЦИОННЫЕ БАЗЫ ДАННЫХ

В ПРИМЕРАХ

С. С. КУЛИКОВ

РЕЛЯЦИОННЫЕ БАЗЫ ДАННЫХ В ПРИМЕРАХ

ПРАКТИЧЕСКОЕ ПОСОБИЕ ДЛЯ ПРОГРАММИСТОВ И ТЕСТИРОВЩИКОВ

МИНСК
ИЗДАТЕЛЬСТВО «ЧЕТЫРЕ ЧЕТВЕРТИ»
2020

УДК 004.652.4
ББК 32.973
К90

Куликов, С. С.
К90 Реляционные базы данных в примерах : практическое пособие для программистов и тестировщиков / С. С. Куликов. — Минск: Четыре четверти, 2020. — 424 с.
ISBN 978-985-581-406-2.

Все ключевые идеи реляционных СУБД — от понятия данных до логики работы транзакций; фундаментальная теория и наглядная практика проектирования баз данных: таблицы, ключи, связи, нормальные формы, представления, триггеры, хранимые процедуры и многое другое в примерах.

Книга будет полезна тем, кто: когда-то изучал базы данных, но что-то уже забыл; имеет узкий практический опыт, но хочет расширить знания; хочет в предельно сжатые сроки начать использовать реляционные базы данных в своей работе.

УДК 004.652.4
ББК 32.973

ISBN 978-985-581-406-2

СОДЕРЖАНИЕ

ПРЕДИСЛОВИЕ

Эта книга посвящена практическому взгляду на реляционную теорию и проектирование реляционных баз данных. Здесь не рассматриваются такие фундаментальные основы, как реляционная алгебра и реляционное счисление, но с множеством примеров и пояснений показаны основные понятия и подходы, необходимые для проектирования баз данных.

Этот материал в первую очередь будет полезен тем, кто:

- никогда не изучал базы данных;
- когда-то изучал базы данных, но многое забыл;
- хочет систематизировать имеющиеся знания.

Все схемы баз данных в этой книге приведены в нотации UML 2.1, созданы с использованием Sparx Enterprise Architect и (если речь идёт об уровнях проектирования, для которых это актуально) ориентированы на MySQL 8.0, Microsoft SQL Server 2019, Oracle 18c. Скорее всего, приведённые решения будут успешно работать на более новых версиях этих СУБД, но **не на более старых**.

Исходные материалы (схемы, скрипты и т.д.) можно получить по этой ссылке:
http://svyatoslav.biz/relational_databases_book_download/src_rdb.zip

Условные обозначения, используемые в этой книге:

Определения и иная важная для запоминания информация. Большинство определений будут приведены пусть и в адаптированной, но достаточно строгой форме.

Для облегчения понимания в каждом определении будет приведена его упрощённая форма (иногда упрощение будет граничить с некорректностью, потому всё же стоит ориентироваться на более строгую формулировку, а упрощённую использовать лишь как подсказку).

Дополнительные сведения или отсылка к соответствующим источникам. Всё то, что полезно знать. При этом оригинальные (англоязычные) определения будут приведены в сносках.

Предостережения и частые ошибки. Недостаточно показать, «как правильно», часто большую пользу приносят примеры того, как поступать не стоит.

Задания для самостоятельной проработки. Настоятельно рекомендуется выполнять их (даже если вам кажется, что всё очень просто; и особенно — если задание выглядит сложным и непонятным).

Материал книги построен таким образом, что его можно как изучать последовательно, так и использовать как быстрый справочник (на все необходимые пояснения в тексте даны ссылки).

Приступим!

ОСНОВЫ БАЗ ДАННЫХ

ДАННЫЕ И БАЗЫ ДАННЫХ

Традиционно начнём с определений, которые едва ли откроют для вас что-то новое, но позволят нам в дальнейшем избежать многократных пояснений одних и тех же мыслей.

Данные (data[1]) — поддающееся различной интерпретации представление информации в формализованном виде, пригодном для передачи, связи, или обработки.

Упрощённо: информация, организованная по определённым правилам.

Ключевых моментов здесь два.

Во-первых, информация должна быть представлена в формализованном виде (иными словами, подчиняться неким правилам). Для наглядности приведём пример неформализованного и формализованного представления информации:

Неформализованное представление	Формализованное представление
«У нас в отделе работают Иванов и Петров, и у Петрова телефон недавно поменялся, был 123-44-55, стал 999-87-32. А ещё Сидоров выйдет на работу в следующем месяце, но дозвониться до него тяжело будет, т.к. он на 333-55-66 не часто сидит, а чаще там другой Сидоров отвечает, но он из другого отдела.»	**employee** **pass** | **name** | **department** | **phone** 178 | Иванов И.И. | Отд-1 | NULL 223 | Петров П.П. | Отд-1 | 999-87-32 243 | Сидоров С.С. | Отд-1 | 333-55-66 318 | Сидоров С.С. | Отд-2 | 333-55-66

Во-вторых, различная интерпретация позволяет нам по-разному воспринимать одни и те же данные в разном контексте. Например, поле **`pass`** (пропуск) может быть интерпретировано так:

- номер пропуска сотрудника;
- поле с уникальным значением;
- число;
- естественный первичный ключ отношения;
- кластерный индекс{110} таблицы;
- и т.д.

[1] **Data** — reinterpretable representation of information in a formalized manner suitable for communication, interpretation, or processing (ISO/IEC 2382:2015, Information technology — Vocabulary).

Когда данных становится достаточно много, появляется необходимость в их организации в более сложные структуры.

База данных (database[2]) — совокупность данных, организованных в соответствии с концептуальной структурой, описывающей характеристики этих данных и взаимоотношения между соответствующими сущностями и поддерживающей одну или более областей применения.

Упрощённо: большой объём данных, взаимосвязь между которыми построена по специальным правилам.

Итак, база данных — это большая совокупность данных, подчинённых дополнительным правилам. Такие правила зависят от вида базы данных, а за их соблюдение отвечает система управления базами данных (СУБД).

Система управления базами данных, СУБД (database management system[3], DBMS) — система (базирующаяся на программном и аппаратном обеспечении) для описания, создания, использования, контроля и управления базами данных.

Упрощённо: программное средство, управляющее базами данных.

Стоит отметить важный для начинающих факт (т.к. очень часто можно услышать просьбу «показать СУБД»): подавляющее большинство СУБД не имеет никакого «человеческого интерфейса», представляет собой сервис (демон в *nix-системах) и взаимодействует с внешним миром по специальным протоколам (чаще всего, построенным поверх TCP/IP). Такие известные продукты как MySQL Workbench, Microsoft SQL Server Management Studio, Oracle SQL Developer и им подобные — это **не** СУБД, это лишь клиентское программное обеспечение, позволяющее нам взаимодействовать с СУБД.

Итак, СУБД — это специальное программное обеспечение, предназначенное для управления базами данных. Поскольку в этой книге мы будем говорить исключительно о реляционных базах данных и реляционных же СУБД, можно сразу отметить, что взаимодействие с такими СУБД происходит путём выполнения SQL-запросов.

Изучение языка SQL и его диалектов выходит за рамки данной книги, но если вы знакомы с общими концепциями, можно расширить свои знания, ознакомившись с материалом книги[4] «Работа с MySQL, MS SQL Server и Oracle в примерах».

Задание 1.1.a: для расширения кругозора найдите самостоятельно информацию о том, каким образом происходит взаимодействие клиентских приложений с нереляционными СУБД, т.е. что в таких СУБД «заменяет» собой SQL-запросы.

Задание 1.1.b: на основе информации, представленной в приложении **1**{408}, а также на основе раздаточного материала{5} сформируйте даталогическую модель базы данных «Банк» для MySQL. Если на текущий момент это задание окажется слишком сложным, отложите его на некоторое время и продолжите изучение материала книги.

Задание 1.1.c: на основе информации, представленной в приложении **2**{412}, а также на основе раздаточного материала{5} сформируйте физическую модель базы данных «Банк» для MySQL. Если на текущий момент это задание окажется слишком сложным, отложите его на некоторое время и продолжите изучение материала книги.

---

[2] **Database** — collection of data organized according to a conceptual structure describing the characteristics of these data and the relationships among their corresponding entities, supporting one or more application areas (ISO/IEC 2382:2015, Information technology — Vocabulary).

[3] **Database Management System, DBMS** — system, based on hardware and software, for defining, creating, manipulating, controlling, managing, and using databases (ISO/IEC 2382:2015, Information technology — Vocabulary).

[4] «Работа с MySQL, MS SQL Server и Oracle в примерах» (С.С. Куликов) [http://svyatoslav.biz/database_book/]

1.2. МОДЕЛИРОВАНИЕ БАЗ ДАННЫХ

Традиционно этот раздел в большинстве книг, посвящённых базам данных, пролистывают, не читая. Прошу на этот раз всё же обратить на него внимание — будет максимально кратко и по сути. И поможет лучше понять, откуда взялись те или иные идеи в следующих разделах.

Модель базы данных (database model, data model[5]) — способ описания базы данных с помощью формализованного (в т.ч. графического) языка на некотором уровне абстракции.

Упрощённо: описание базы данных, по которому она потом будет создана (по аналогии с проектом здания, по которому оно потом будет построено).

Особый интерес в данном определении представляет упоминание уровня абстракции. Получается, что у одной и той же базы данных может быть несколько моделей, отличающихся уровнем детализации и целью (например, общее описание данных и их взаимосвязей, описание структур данных, описание способов хранения и обработки данных и т.д.)

Отсюда мы переходим к понятию уровней (этапов) моделирования и проектирования баз данных, т.е. к построению нескольких взаимосвязанных моделей базы данных, каждая из которых призвана служить своей особой цели.

Вы можете найти много различных (иногда — противоречащих друг другу) классификаций уровней моделирования баз данных. Такая неразбериха обусловлена тем, что автор каждой классификации основывался на взгляде, актуальном для своего времени и своих технологий и по-своему связанном с общими вопросами моделирования информационных систем. Нижеприведённая классификация не является исключением, и тоже подвержена искажениям, связанным с точкой зрения автора.

Единственное, что нерушимо объединяет все классификации — это идея возможности нисходящего{11} и восходящего{12} проектирования, а также тот факт, что с каждым следующим, более низким уровнем моделирования возрастает детализация модели и усиливается её привязка к СУБД конкретного типа, конкретного производителя, конкретной версии и т.д.

Также стоит отметить, что во многих случаях тяжело чётко обозначить границу между уровнями, то есть они плавно перетекают друг в друга (как минимум в момент соответствующих подготовительных действий в процессе проектирования).

Прежде, чем мы рассмотрим каждый уровень (см. схему на рисунке 1.2.а), необходимо дать ещё одно предельно важное пояснение, без которого вся эта информация может оказаться бесполезной.

Как правило, студенты технических вузов, изучающие базы данных и проектирующие их в рамках лабораторных, курсовых и дипломных работ, искренне не понимают, зачем нужны какие-то уровни, если можно просто «взять и спроектировать базу».

Можно. В том случае, если база примитивная (до нескольких десятков таблиц), предметная область ограничивается уровнем сложности университетской работы, исходные требования

---

[5] **Data model** — pattern of structuring data in a database according to the formal descriptions in its information system and according to the requirements of the database management system to be applied (ISO/IEC 2382:2015, Information technology — Vocabulary).

просты и не меняются. Т.е. если вы проектируете однотипные и/или простые базы данных, вся эта «научная заумь» вам не понадобится. (Особенно, если вы сами и заказчик, и исполнитель, и ничто не мешает опустить руки и поменять требования, как только возникнет достаточно сложная задача.)

Но! Представьте, что предметная область вам совершенно неизвестна (например, вы проектируете не очередную базу данных библиотеки, а информационное хранилище для исследовательского центра кардиохирургии).

Добавьте к этому тот факт, что сам заказчик (в лице представителей этого центра) не сильно разбирается в информационных технологиях, и многое объясняет вам «на пальцах», причём в процессе работы многократно выясняется, что что-то было забыто, что-то вы не так поняли, что-то поменялось, чего-то не знал сам заказчик и т.д.

Причём в этой ситуации, как правило, не удастся отказаться от реализации того или иного требования просто потому, что оно слишком сложное и/или вы не очень понимаете, как его реализовать.

Также добавьте к этому тот факт, что база данных может состоять из сотен (и даже тысяч) таблиц, что к ней предъявляются достаточно жёсткие требования по надёжности, производительности и иным показателям качества.

Учтите, что проектировать такую базу вы будете не в одиночку, а в составе команды (и всем участникам надо будет гарантированно понимать друг друга, согласовывать свои действия, подменять друг друга).

Из соображений оптимизма остановим это перечисление (хотя его можно продолжить на несколько страниц).

В такой ситуации структурированный, управляемый, регламентированный подход к моделированию и проектированию критически необходим — без него создать работоспособную базу данных не получится. Никак.

Возвращаемся к уровням моделирования.

*Рисунок 1.2.а — **Обобщённая схема уровней моделирования базы данных***

Инфологический (концептуальный) уровень (conceptual level[6]) моделирования ставит своей целью создание т.н. концептуальной модели (conceptual model[7]), отражающей основные сущности предметной области, их атрибуты и связи (возможно, пока не все) между сущностями.

Упрощённо: описание предметов и явлений реального мира, данные о которых потом будут помещены в базу данных.

Если продлить рисунок 1.2.а «вверх», к ещё меньшему уровню детализации, от непосредственно работы с базами данных мы перейдём к области бизнес-анализа (business analysis), выявления общих требований заказчика и параметров предметной области.

Очень подробно и наглядно весь соответствующий материал изложен в книге Карла Вигерса «Разработка требований к программному обеспечению» (Karl E. Wiegers, «Software Requirements»).

Как предметную область целиком, так и непосредственно данные на этом уровне можно описывать различными моделями (например, семантической, графовой, «сущность-связь»). В качестве способа представления модели чаще всего используется UML или простое словесное описание (что особенно удобно для обсуждения модели с представителем заказчика, не являющимся IT-специалистом).

Подробности реализации этой задачи будут рассмотрены в главе «Проектирование на инфологическом уровне{284}».

Даталогический уровень (часто его называют просто «логическим», logical level[8]) моделирования детализирует инфологическую модель, превращая её в логическую схему (logical schema[9]), на которой ранее выявленные сущности, атрибуты и связи оформляются согласно правилам моделирования для выбранного вида базы данных (возможно, даже с учётом конкретной СУБД).

Упрощённо: описание предметов и явлений реального мира по правилам выбранной СУБД.

Во многих средствах проектирования баз данных, поддерживающих разделение лишь на «логическое» и «физическое» проектирование именно этот уровень называется «логическим».

Здесь уже появляются вполне привычные таблицы{21}, поля{21}, ключи{35}, связи{57}, часть индексов{106} и представлений{340} — всё то, что и составляет суть базы данных. Поскольку реализация этих объектов зависит от вида базы данных (и даже конкретной СУБД), здесь уже нельзя строить строго абстрактную модель, и приходится учитывать типичные особенности выбранных базы данных и СУБД.

В качестве способа представления модели на этом уровне чаще всего будет использоваться UML или постепенно утрачивающие популярность нотация IDEF1X, нотация Чена и им подобные.

Подробности реализации этой задачи будут рассмотрены в главе «Проектирование на даталогическом уровне{304}».

---

6 **Conceptual level** — level of consideration at which all aspects deal with the interpretation and manipulation of information describing a particular universe of discourse or entity world in an information system (ISO/IEC 2382:2015, Information technology — Vocabulary).

7 **Conceptual model** — representation of the characteristics of a universe of discourse by means of entities and entity relationships (ISO/IEC 2382:2015, Information technology — Vocabulary).

8 **Logical level** — level of consideration at which all aspects deal with a database and its architecture, consistent with a conceptual schema and the corresponding information base, but abstract from its physical implementation (ISO/IEC 2382:2015, Information technology — Vocabulary).

9 **Logical schema** — part of the database schema that pertains to the logical level (ISO/IEC 2382:2015, Information technology — Vocabulary).

Стоит отметить, что даталогических моделей много (документальная, фактографическая, теоретико-графовая, теоретико-множественная, объектно-ориентированная, основанная на инвертированных файлах и т.д.). И каждая из них породила свой вид баз данных (иногда — несколько видов).

Данная книга посвящена реляционным базам данных, но остальные (основные) хотя бы перечислим: картотеки, сетевые базы данных, иерархические базы данных, многомерные базы данных, объектно-ориентированные базы данных, дедуктивные базы данных, NoSQL базы данных и т.д.

Физический уровень (physical level[10]) моделирования продолжает детализацию и позволяет создать т.н. физическую схему (physical schema[11]), на которой максимально учитываются технические особенности работы конкретной СУБД и её возможности по организации и управлению структурами разрабатываемой базы данных и данными в ней.

Упрощённо: описание составных частей базы данных таким образом, чтобы на его основе можно было автоматически сгенерировать SQL-код для создания базы данных.

Если продлить рисунок 1.2.а «вниз», к ещё большему уровню детализации, от непосредственно работы с базами данных мы перейдём к области системного администрирования (конфигурирования СУБД, операционной системы, сетевой инфраструктуры и т.д.)

На этом уровне модель данных может быть представлена так же, как и на предыдущем (даталогическом) — чаще всего, в виде UML, но одной лишь графической формы здесь недостаточно, потому в ход идут SQL-скрипты, словесные описания необходимых изменений и настроек, фрагменты конфигурационных файлов, подготовленные cmd/bash-скрипты, reg-файлы и т.д.

Подробности реализации этой задачи будут рассмотрены в главе «Проектирование на физическом уровне{314}».

Ранее было упомянуто, что любая классификация уровней моделирования допускает как нисходящее, так и восходящее проектирование. Поясним подробнее.

Нисходящее проектирование (top-down[12] design) — в контексте проектирования баз данных часто называется «проектированием от предметной области», предполагает движения от самого высокого уровня моделирования (инфологического) вниз к самому низкому (физическому).

Упрощённо: начинаем общаться с заказчиком, а потом думаем, как реализовать его требования.

---

10 **Physical level** — level of consideration at which all aspects deal with the physical representation of data structures and with mapping them on corresponding storage organizations and their access operations in a data processing system (ISO/IEC 2382:2015, Information technology — Vocabulary).

11 **Physical schema** — part of the database schema that pertains to the physical level (ISO/IEC 2382:2015, Information technology — Vocabulary).

12 **Top-down** — pertaining to a method or procedure that starts at the highest level of abstraction and proceeds towards the lowest level (ISO/IEC 2382:2015, Information technology — Vocabulary).

Восходящее проектирование (bottom-up[13] design) — в контексте проектирования баз данных часто называется «проектированием от запросов», предполагает движения от самого низкого уровня моделирования (физического) вверх к самому высокому (инфологическому).

Упрощённо: смотрим, что и как у нас получается реализовать, и потом думаем, как с помощью этого выполнить требования заказчика.

На практике, чтобы получившаяся база данных удовлетворяла всем ключевым требованиям (будут рассмотрены очень скоро{12}), оба направления проектирования применяются попеременно. Отсюда следует, что процесс проектирования является итерационным, и невозможно «сделать и забыть» модель на каком-то уровне, полностью переключившись на другой.

Если попытаться совершить такую ошибку, выбрав строго одно из направлений проектирования (да ещё и не пересматривая пройденный путь с целью обнаружения и исправления ошибок), почти гарантированно будет нарушено одно из следующих требований, предъявляемых к любой базе данных (и, соответственно, её моделям):

- адекватность предметной области;
- удобство использования;
- производительность;
- защищённость данных.

Канонических определений здесь не существует, но сами идеи крайне важны, потому мы рассмотрим их подробно и с примерами.

С более строгой трактовкой некоторых из рассматриваемых здесь требований можно ознакомиться в книге «Основы баз данных» (Кузнецов С.Д.)

Адекватность предметной области выражается в том, что база данных должна позволять выполнять все необходимые операции, которые объективно нужны в реальной жизни в контексте той работы, для которой предназначена база данных.

Лучше всего выполнить это требование позволяет моделирование на инфологическом уровне (т.к. этот уровень ближе всего находится к предметной области и требованиям заказчика), но для выявления некоторых ошибок неизбежно придётся анализировать и модели на других уровнях. А суть этого требования наиболее наглядно отражают примеры его нарушения.

Сначала рассмотрим совершенно тривиальный пример. Допустим, для хранения информации о человеке было спроектировано отношение, представленное следующей схемой (рисунок 1.2.b):

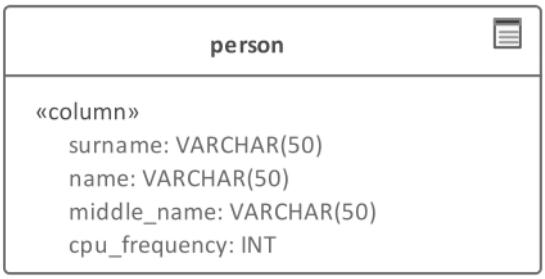


*Рисунок 1.2.b — **Тривиальный пример нарушения адекватности предметной области***

[13] **Bottom-up** — pertaining to a method or procedure that starts at the lowest level of abstraction and proceeds towards the highest level (ISO/IEC 2382:2015, Information technology — Vocabulary).

Сразу же бросается в глаза, что `cpu_frequency` (частота процессора) не является характеристикой человека (по крайней мере, на сегодняшний день). А потому совершенно непонятно, что писать в это поле.

Если немного подумать, также становится очевидным, что в этом отношении не хватает многих свойств, ведь характеристики человека не заканчиваются на фамилии, имени и отчестве.

Рассмотрим чуть более близкий к реальности пример ошибки проектирования (здесь даже обойдёмся без рисунка). Допустим, мы разрабатываем базу данных для автоматизации работы деканата университета. Соответствующее приложение уже введено в эксплуатацию, прошло какое-то время, и тут нам звонят заказчики и спрашивают: «А как отметить, что студент ушёл в академический отпуск?»

И мы вдруг понимаем, что… никак. Что мы не подумали о такой ситуации, и ни приложение (с которым работают сотрудники деканата), ни наша база данных не имеют возможности сохранить соответствующие данные.

Эта ситуация является неприятной (придётся много переделывать), но она меркнет по сравнению со следующей — худшей из того, что может случиться.

Рассмотрим третий (по-настоящему страшный) пример, продолжив тему автоматизации работы деканата. Допустим, что информация о взаимосвязи предметов, преподавателей и студентов хранилась в базе данных, представленной (упрощённо) следующей схемой (рисунок 1.2.c):

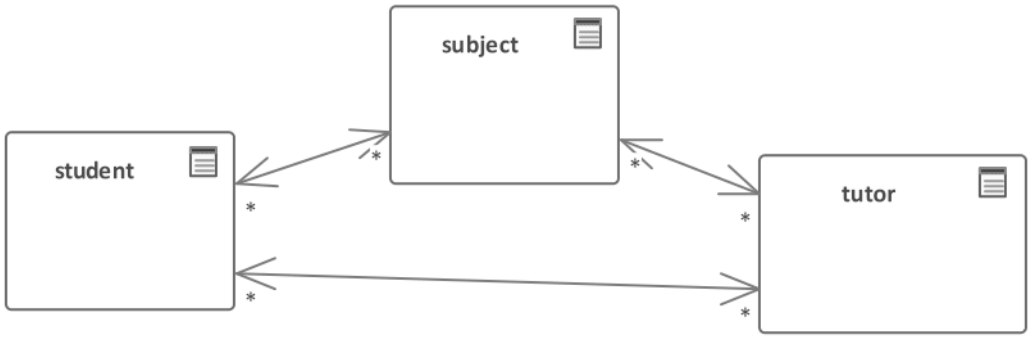


*Рисунок 1.2.c — **Упрощённая схема базы данных, в которой скрыта ошибка***

Все три отношения попарно соединены связями «многие ко многим{60}», ведь действительно:

- каждый студент изучает много предметов, и каждый предмет изучает много студентов;
- каждый предмет может вести много преподавателей, и каждый преподаватель может вести много предметов;
- каждый студент обучается у многих преподавателей, и каждый преподаватель обучает многих студентов.

Допустим, разработанный продукт успешно введён в эксплуатацию, прошло много лет, и тут заказчик звонит и интересуется, как для очень важного отчёта показать список студентов, у которых профессор Иванов вёл высшую математику.

Никак. Эта информация не сохранялась в базе данных. Там есть лишь информация о том, что профессор Иванов (вместе с ещё тремя десятками коллег) вёл высшую математику (и ещё восемь предметов). Есть информация о том, какие студенты учились у профессора Иванова, и какие студенты изучали высшую математику. Но никак невозможно определить, что вот именно этот конкретный студент изучал именно высшую математику (а не другой предмет) именно у профессора Иванова (а не у кого-то из его коллег). (Альтернативная схема, устраняющая данную проблему, будет рассмотрена далее{167}.)

Этот пример потому и назван самым страшным, что тут ничего нельзя исправить. Даже если мы доработаем приложение и базу данных, требуемая заказчиком информация всё равно останется утерянной навсегда. А ведь заказчик был полностью уверен, что она есть и доступна.

В принципе, при грамотном подходе к сбору требований такие ошибки (как в примерах про деканат) выявляются и при нисходящем проектировании, но намного легче заметить их при восходящем проектировании, когда мы задумываемся о том, какие конкретно задачи будет решать пользователь, и к выполнению каких запросов к базе данных это приведёт.

Существует ещё одна форма нарушения адекватности предметной области — нарушения нормализации, которые приводят к возникновению аномалий операций с данными{161}. Но этот вопрос настолько важен и обширен, что ему посвящён отдельный раздел{161}.

Удобство использования (в контексте проектирования баз данных) не связано с удобством для конечного пользователя (который может даже не знать, что в мире существуют какие-то базы данных). Этот термин относится к использованию базы данных в следующих ситуациях:

- при взаимодействии с приложениями (в основном, здесь упор делается на простоту и лёгкость написания безошибочных запросов);
- в процессе её дальнейшего развития (и тогда говорят о таких показателях качества, как поддерживаемость и сопровождаемость).

Таким образом, речь идёт об «удобстве для программиста». Почему это важно? Чем лучше база данных отвечает данному требованию, тем проще и быстрее программисты могут организовать взаимодействие с ней, допуская минимум ошибок и с меньшими затратами решая вопросы быстродействия, безопасности и т.д.

Прекрасной иллюстрацией разницы в удобстве использования базы данных в зависимости от её модели, может быть пример с определением номера дня недели и номера недели в месяце на основе указанной даты (см. очень подробное описание этой задачи с решением в соответствующей книге[14]).

Пока дата хранится в виде единого значения, приходится использовать запрос на целый экран, причём для каждой СУБД в нём появляется своя «магия» в виде числовых констант, слабо документированных параметров функций и тому подобных крайне опасных с точки зрения возможности допустить ошибку моментов.

Но модель базы данных можно изменить. Можно хранить искомые значения в самой таблице (и вычислять их триггерами{350}) или создать представление{340} (и «замаскировать» соответствующий запрос в нём). Тогда для программиста, организующего взаимодействие приложения с базой данных задача сведётся к тривиальному SELECT длиной в пару строк (схематично идея показана на рисунке 1.2.d).

---

[14] «Работа с MySQL, MS SQL Server и Oracle в примерах» (С.С. Куликов), задача 7.4.1.a [http://svyatoslav.biz/database_book/]

MS SQL **Запрос до правки модели**

```
WITH [iso_week_data] AS
 (SELECT CASE
          WHEN DATEPART(iso_week,
               CONVERT(VARCHAR(6), [sb_start], 112) + '01') >
               DATEPART(dy,
               CONVERT(VARCHAR(6), [sb_start], 112) + '01')
          THEN 0
          ELSE DATEPART(iso_week,
               CONVERT(VARCHAR(6), [sb_start], 112) + '01')
         END AS [real_iso_week_of_month_start],
         CASE
          WHEN DATEPART(iso_week, [sb_start]) >
               DATEPART(dy, [sb_start])
          THEN 0
          ELSE DATEPART(iso_week, [sb_start])
         END AS [real_iso_week_of_this_date],
         [sb_start], [sb_id]
  FROM [subscriptions])
 SELECT [sb_id], [sb_start],
        CASE
         WHEN DATEPART(dw,
              DATEADD(day, -1, CONVERT(VARCHAR(6), [sb_start], 112)
              + '01')) <= DATEPART(dw, DATEADD(day, -1, [sb_start]))
         THEN [real_iso_week_of_this_date] -
              [real_iso_week_of_month_start] + 1
         ELSE [real_iso_week_of_this_date] -
              [real_iso_week_of_month_start]
        END AS [W],
        DATEPART(dw, DATEADD(day, -1, [sb_start])) AS [D]
FROM [iso_week_data]
```

MS SQL **Запрос после правки модели**

```
SELECT [sb_id],
       [sb_start],
       [sb_start_week]
       [sb_start_day]
FROM   [subscriptions]
```

Рассмотрим ещё один очень простой и наглядный пример нарушения удобства использования. Предположим, что в некоторой базе данных фамилии и инициалы людей сохранены следующим образом (см. рисунок 1.2.е).

person

p_id	p_name	...
1	И.И. Иванов	...
2	ПП Петров	...
3	С Сидоров	...
4	...	...

*Рисунок 1.2.е — **Неоптимальное хранение данных***

Здесь фамилия и инициалы сохранены в одной колонке, причём инициалы идут перед фамилией (и для наглядности представлены в нескольких вариантах, что вполне может встретиться в реальной жизни). Как теперь упорядочить список людей по алфавиту по фамилии?

Теоретически можно написать функцию, учитывающую множество вариантов написания инициалов (например, «А.А.», «АА», «А. А. », «А .» и т.д. и т.п.), убирающую эту информацию и возвращающую только фамилию. И использовать эту функцию в запросах следующего вида:

MySQL **Упорядочивание списка людей по фамилии по алфавиту с удалением инициалов**

```
SELECT *
  FROM `person`
ORDER BY REMOVE_INITIALS(`p_name`) ASC
```

Но такая функция будет сложной, медленной, может не учитывать некоторые варианты написания инициалов, а также сделает невозможным использование индексов{106} для ускорения выполнения запроса (или потребует создания отдельного индекса над результатами своих вычислений).

Но всего лишь стоит поместить фамилии и инициалы в раздельные колонки (см. рисунок 1.2.f), и ситуация мгновенно упрощается.

person

p_id	p_surname	p_initials	...
1	Иванов	И.И.	...
2	Петров	ПП	...
3	Сидоров	С	...
4	...	...	...

*Рисунок 1.2.f — **Оптимальное хранение данных***

Теперь нет необходимости как бы то ни было предварительно обрабатывать информацию, и значение фамилии можно использовать напрямую, что устраняет все недостатки решения с использованием функции. И запрос упрощается до следующего вида:

MySQL **Упорядочивание списка людей по фамилии по алфавиту с удалением инициалов**

```
1    SELECT *
2      FROM `person`
3  ORDER BY `p_surname` ASC
```

Подобные ситуации могут встречаться чаще, чем кажется. И их продумывание позволяет ощутимо повысить удобство использования базы данных.

Производительность можно смело считать одним из самых больных вопросов в работе с базами данных, и решение соответствующих задач может быть описано несколькими отдельными книгами. Что стоит делать на любой стадии проектирования — так это как минимум помнить о производительности и не допускать создания таких моделей, в которых на ней будет поставлен крест изначально.

Например, мы знаем, что для работы приложения понадобится очень часто выяснять количество записей в разных таблицах (как общее, так и количество неких особенных записей). В общем случае такие операции будут работать очень медленно (см. подробное описание этой ситуации в соответствующей книге[15]), но с использованием кэширующих таблиц{235} и материализованных представлений{340} можно ценой небольшой потери производительности других операций в этом конкретном случае свести время выполнения запроса почти к нулю.

Как правило, проблемы с удобством использования{14} приводят к увеличению количества проблем с производительностью, т.к. сложные запросы тяжелее оптимизировать.

---

[15] «Работа с MySQL, MS SQL Server и Oracle в примерах» (С.С. Куликов), задача 2.1.3.a [http://svyatoslav.biz/database_book/]

В общем же случае, невозможно без всестороннего анализа создать модель базы данных, которая гарантированно обеспечивала бы высокую производительность — здесь роль играет слишком много факторов, часть из которых вообще выходит за рамки проектирования базы данных (например, структура дискового хранилища).

Но повторим ещё раз самое важное: для начала стоит избегать как минимум самых очевидных глупых ошибок, например (сознательно утрируем), создания искусственного первичного ключа размером 2GB в таблице, хранящей названия дней недели.

Защищённость данных стоит воспринимать не только в контексте безопасности (ограничения доступа), но и в том смысле, что с данными не должно происходить никаких случайных непредвиденных изменений. Для достижения этой цели порой приходится продумывать огромное количество ограничений, реализуемых через специфические объекты базы данных и даже через отдельный интерфейс в виде набора хранимых процедур{363}.

В качестве примера упомянем классику — хранение в реляционной базе данных дерева (см. схему на рисунке 1.2.g).

На первый взгляд всё красиво и логично, но для того, чтобы обеспечить гарантированную корректность выполнения всех операций (добавление вершины в произвольное место, удаление вершины, перенос и/или копирование поддерева и т.д.) и соблюдения всех правил (строго один корень, отсутствие обрывов, колец и т.д.) приходится создавать десятки дополнительных объектов базы данных и запрещать прямое изменение данных в соответствующей таблице. Без всех этих мер в любой момент времени структура дерева может быть нарушена.

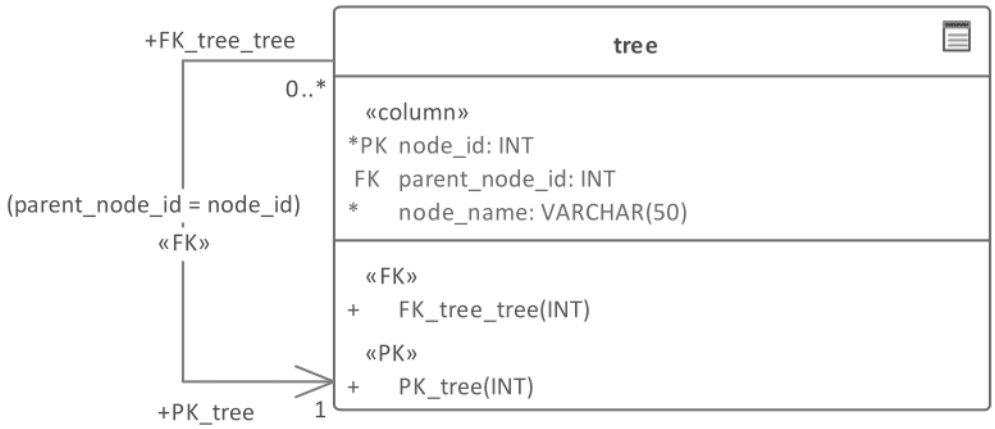


*Рисунок 1.2.g — **Классическая схема для хранения дерева***

Здесь снова упомянем аномалии операций с данными{161} как ещё один яркий пример нарушения защищённости данных.

В завершении данной главы ещё раз подчеркнём, что для обеспечения соответствия базы данных всем рассмотренным выше требованиям необходим тщательный процесс моделирования, в котором не стоит полагаться на удачу. И результаты такого моделирования должны быть тщательно документированы.

Задание 1.2.a: для лучшего понимания приведённых в данной книге UML-диаграмм ознакомьтесь с хотя бы базовыми принципами UML. Много хороших кратких справочных материалов можно найти здесь:
http://modeling-languages.com/best-uml-cheatsheets-and-reference-guides/

Задание 1.2.b: приведите несколько примеров моделей баз данных, в которых были бы нарушены ключевые требования к базам данных{12}.

Задание 1.2.c: какие недоработки в инфологической и даталогической моделях базы данных «Банк»{408} вы видите? Внесите соответствующие правки в эти модели.

1.3. РЕЛЯЦИОННАЯ МОДЕЛЬ ДАННЫХ

Пожалуй, эта глава — самая теоретическая во всей книге. Но здесь мы рассмотрим те самые важные свойства реляционной модели, которые вот уже более половины столетия (модель предложена в 1969-м году) позволяют ей занимать лидирующие места по множеству показателей.

Реляционная модель (relational model[16]) — математическая теория, описывающая структуры данных, логику контроля целостности данных и правила управления данными.

Упрощённо: модель для описания реляционных баз данных.

На момент изобретения именно эта модель предоставила полный комплекс решений по описанию базы данных, контролю целостности данных и управлению данными — в отличие от множества других альтернативных подходов, позволяющих реализовать лишь часть задач. Сейчас кажется странным, что были какие-то другие «неполноценные решения», но напомним: это было в те времена, когда компьютер занимал несколько комнат и стоил, как половина здания, в котором находится.

В контексте описания структур данных реляционной моделью вводится понятие нормализованного{161} отношения{21}. Более того, нормализованное отношение является единственной базовой структурой, которой оперирует эта модель.

В контексте целостности данных речь идёт о целостности сущностей (таблиц, отношений {21}) и внешних ключей{35}.

И, наконец, в контексте правил управления данными речь идёт о реляционной алгебре и реляционном счислении. При всей красоте этих явлений, их рассмотрение, увы, выходит за рамки данной книги.

Очень подробное описание реляционной алгебры и реляционного счисления приведено в книге «Основы баз данных» (Кузнецов С.Д.)

Почему реляционная модель столь удобна? В силу следующих достоинств:

- она является логической (а не физической, т.к. физическая реализация отдана разработчикам СУБД), что позволяет на достаточно высоком уровне абстракции проектировать базу данных, не привязываясь к конкретной физической реализации таблиц, связей и прочих объектов (и в итоге, например, выбирать конкретную СУБД уже после построения инфологической модели);
- она построена на основе строгого математического аппарата, что позволяет не только определять применимость, принципиальную возможность и корректность тех или иных операций, но и оценивать их сложность (и даже служить основой для расчётов показателей производительности);
- она описывает как декларативный подход (программист указывает конечную цель решения без описания последовательности действий), так и процедурный (знакомый всем по «классическому программированию», т.е. представляющий собой описание последовательности шагов для достижения поставленной цели).

---

16 **Relational model** — data model whose structure is based on a set of relations (ISO/IEC 2382:2015, Information technology — Vocabulary).

Декларативный подход (в виде языка SQL) реализован почти идентично в подавляющем большинстве реляционных СУБД, в то время как процедурный (триггеры, хранимые процедуры и функции) имеет столько особенностей, что можно смело говорить о полной несовместимости соответствующих решений в разных СУБД.

«На бытовом» уровне главным достоинством реляционной модели стала её повсеместная распространённость: существует много хорошо зарекомендовавших себя СУБД, много готовых проверенных решений, много квалифицированных специалистов, много литературы и т.д.

Из недостатков реляционной модели стоит отметить, что:

- она проигрывает другим моделям в оптимальности описания и использования некоторых структур данных (например, реализация деревьев или графов в иерархической или графовой модели оказывается на порядки проще);
- она не избавлена от типичной и для других моделей проблемы высокой сложности (для человека) разработки больших баз данных (хотя частично это компенсируется возможностями средств проектирования);
- построенные на основе реляционной модели базы данных при выполнении многих операций являются весьма требовательными к объёмам памяти и мощности вычислительных ресурсов (именно этот недостаток стал одной из предпосылок появления NoSQL-решений).

И всё же до сих пор у реляционной модели нет альтернативы, которая сочетала бы в себе все её достоинства и универсальность и была бы избавлена от её недостатков. Потому для подавляющего большинства случаев, в которых для хранения и обработки данных нужна база данных, речь автоматически идёт именно о реляционных базах данных.

На этом вся общая теория закончена, и дальше мы будем рассматривать фундаментальные понятия реляционных баз данных с максимальным количеством практических примеров.

Если вас всё же заинтересовала научная часть, ознакомьтесь с этими книгами:

- «Основы баз данных» (Кузнецов С.Д.)
- «Введение в реляционные базы данных» (Кириллов В.В., Громов Г.Ю.)
- «Введение в системы баз данных» (К. Дж. Дейт, 8-е издание)

Задание 1.3.a: назовите ещё по пять преимуществ и недостатков реляционной модели (поищите соответствующую информацию в упомянутых выше книгах).

Задание 1.3.b: можно ли представить модель базы данных «Банк»{408} не в реляционной форме, а в какой бы то ни было другой? Попробуйте изобразить альтернативное решение графически в любой удобной вам форме.

Задание 1.3.c: как вы считаете, почему для описания базы данных «Банк»{408} была использована именно реляционная модель? В чём преимущество такого решения в сравнении с доступными альтернативами?

ОТНОШЕНИЯ, КЛЮЧИ, СВЯЗИ, ИНДЕКСЫ

2.1. ОТНОШЕНИЯ

2.1.1. ОБЩИЕ СВЕДЕНИЯ ОБ ОТНОШЕНИЯХ

Ранее{19} мы отметили, что нормализованное{161} отношение является единственной базовой структурой, которой оперирует реляционная модель данных. Однако с понятием «отношение» неразрывно связаны такие термины как «тип данных», «домен данных», «атрибут», «кортеж», «первичный ключ{39}», большую часть которых мы последовательно рассмотрим в данной главе. И начнём с основного термина.

Отношение (relation[17]) — множество сущностей, обладающих одинаковым набором атрибутов. В контексте реляционных баз данных отношение состоит из заголовка (схемы) и тела (набора кортежей).

Упрощённо: математическая модель таблицы базы данных.

Важно разделять понятия «отношение» (relation) и «связь» (relationship), несмотря на то, что очень часто в русскоязычной (переводной) литературе их путают. Строго говоря, термин (например) «отношение многие ко многим» неверен, должно быть «связь многие ко многим».

В свете только что сделанного примечания необходимо пояснить ещё одну типичную терминологическую проблему. Сложность в том, что понятие «сущности» крайне расплывчато и часто определяется как «всё, что реально существует в мире». Давайте усугубим ситуацию и переформулируем понятие отношения, итак: «отношение — это сущность, представляющая собой множество сущностей, обладающих определённым набором атрибутов». Уже получается странно.

---

[17] **Relation** — set of entity occurrences that have the same attributes, together with these attributes (ISO/IEC 2382:2015, Information technology — Vocabulary).

Теперь стоит отметить, что и атрибут — это тоже сущность. Итого: «отношение — это сущность, представляющая собой множество сущностей, обладающих определённым набором сущностей». (Можете продолжить преобразование дальше и посмотреть, что получится в итоге.)

Остановим это безумие. В контексте баз данных нас будет волновать только то, что при инфологическом (концептуальном) моделировании, как правило, говорят о «сущностях и связях», а при даталогическом и физическом — об «отношениях и связях» и «таблицах и связях».

Теперь ещё раз, очень кратко и упрощённо: «сущность», «отношение», «таблица» — это синонимы (до тех пор, пока вы не указали точный контекст, в котором используете эти термины). Тогда для упрощения запоминания можно сформулировать более простое определение отношения.

Отношение — множество кортежей (записей, строк таблицы), обладающих одинаковым набором атрибутов (свойств, полей, столбцов таблицы).

Отсюда логично перейти к рассмотрению компонентов отношения. И начнём мы с самого фундаментального — типа данных.

Тип данных (data type[18]) — набор объектов данных определённой структуры и набор допустимых операций, в любой из которых такие объекты могут выступать операндами.

Упрощённо, на примере: числа, строки, даты и т.д.

В общем случае понятие «тип данных» здесь эквивалентно аналогичному понятию в языках программирования (например, «строки» или «числа» трактуются в базах данных и языках программирования почти идентично, а для понимания таких типов данных как «дата-время» или «геометрическая структура» можно представить, что мы имеем дело с экземплярами соответствующих классов).

Все значения, которыми оперирует база данных, являются типизированными (т.е. в любой момент времени известен их тип). Для большинства типов данных существуют операции их преобразования (например, строка «12.34» может быть преобразована в дробное число 12.34).

На типы данных могут налагаться дополнительные ограничения, что приводит к появлению понятия «домен данных».

Домен данных (attribute domain[19]) — набор всех возможных значений атрибута отношения.

Упрощённо, на примере: номер телефона, фамилия, название улицы и т.д.

Суть этого понятия проще всего пояснить на примере. Допустим, в некотором отношении есть атрибут{23} «Код товара», который представлен строкой. И также известно, что все коды товара формируются по правилу «три буквы английского алфавита в верхнем регистре, четыре цифры, две буквы английского алфавита в нижнем регистре» (например, «ABC1234yz»). Очевидно, что строка «понедельник» не является кодом товара, т.е. не входит в соответствующий домен, хоть и относится к тому же типу данных (к строкам).

Понятие домена важно в силу того, что данные считаются сравнимыми только в случае принадлежности к одному домену, т.е. бессмысленно сравнивать код товара и название дня недели (хотя технически такую операцию позволит вам выполнить любая СУБД).

В подавляющем большинстве СУБД приходится использовать триггеры{350} или проверки{347} для контроля принадлежности данных одному домену. Несмотря на некоторую сложность такого

---

18 **Data type** — defined set of data objects of a specified data structure and a set of permissible operations, such that these data objects act as operands in the execution of any one of these operations (ISO/IEC 2382:2015, Information technology — Vocabulary).

19 **Attribute domain** — set of all possible attribute values (ISO/IEC 2382:2015, Information technology — Vocabulary).

контроля и тот факт, что выполнение соответствующих действий снижает производительность некоторых операций с данными, оно позволяет ощутимо повысить защищённость данных{17}, снижая риск передачи (и сохранения) некорректных значений.

Для двух оставшихся терминов приведём определения и сразу перейдём к графическому пояснению.

Атрибут (attribute[20]) — именованное свойство сущности (отношения).

Упрощённо: столбец (колонка) таблицы.

Кортеж (tuple[21]) — часть отношения, представляющая собой уникальную взаимосвязанную комбинацию значений, каждое из которых соответствует своему атрибуту.

Упрощённо: строка (запись) таблицы.

На бытовом уровне под атрибутом можно понимать «столбец таблицы», а под кортежем — «строку таблицы».

Когда в дальнейшем мы будем говорить о нормализации отношений, очень часто будут звучать термины «ключевой атрибут» и «неключевой атрибут». Рассмотрим их.

Ключевой атрибут (key attribute[22], prime attribute[23]) — атрибут отношения, входящий в состав как минимум одного потенциального{37} ключа этого отношения.

Упрощённо: столбец (колонка) таблицы, являющаяся частью потенциального ключа этой таблицы.

Неключевой атрибут (nonkey attribute[24]) — атрибут отношения, не входящий в состав ни одного из потенциальных{37} ключей этого отношения.

Упрощённо: столбец (колонка) таблицы, не являющаяся частью ни одного из потенциальных ключей этой таблицы.

Первичный атрибут (primary key attribute[25]) — атрибут отношения, входящий в состав первичного ключа{39} этого отношения.

Упрощённо: столбец (колонка) таблицы, являющаяся частью первичного ключа этой таблицы.

В русскоязычной литературе очень часто под термином «ключевой атрибут» может пониматься как атрибут, входящий в состав любого потенциального{37} ключа таблицы, так и атрибут входящий в состав первичного{39} ключа таблицы. Соответственно, под термином «неключевой атрибут» может пониматься как атрибут, не входящий в состав ни одного потенциального ключа, так и атрибут, не входящий в состав первичного ключа.

Далее в этой книге термин «ключевой атрибут» будет использован в значении, обозначенном выше{23}.

---

20 **Attribute** — named property of an entity (ISO/IEC 2382:2015, Information technology — Vocabulary).

21 **Tuple** — in a relational database, part of a relation that uniquely describes an entity occurrence and its attributes (ISO/IEC 2382:2015, Information technology — Vocabulary).

22 **Key attribute** — an attribute of a given relvar that's part of at least one key of that relvar. («The New Relational Database Dictionary», C.J. Date)

23 **Prime attribute** — old fashioned and somewhat deprecated term for a key attribute (not necessarily a primary key attribute). («The New Relational Database Dictionary», C.J. Date)

24 **Nonkey attribute** — an attribute of a given relvar that isn't part of any key of that relvar. («The New Relational Database Dictionary», C.J. Date)

25 **Primary key attribute** — an attribute of a given relvar that participates in the primary key (if any) for that relvar. («The New Re-lational Database Dictionary», C.J. Date)

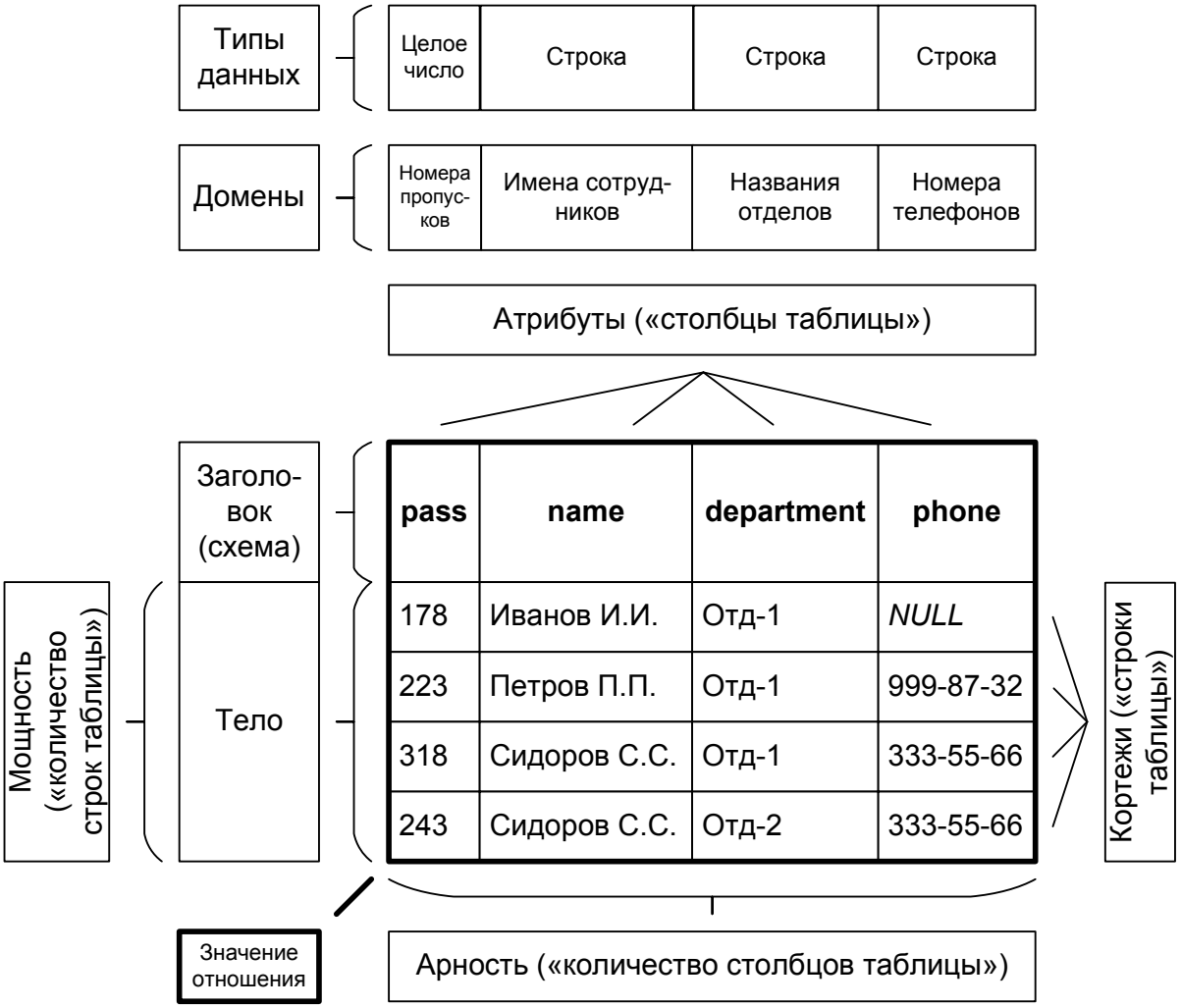


*Рисунок 2.1.а — **Отношение и его составные части***

Важно помнить, что термины «таблица», «столбец», «строка» являются лишь упрощёнными и более удобными в повседневной речи аналогами соответствующих понятий реляционной теории. Но если подходить к вопросу строго математически, такое упрощение будет неверным.

В книге «Введение в системы баз данных» (К. Дж. Дейт, 8-е издание) отношениям и связанным с ними понятиям посвящено в сумме более трёхсот страниц. Если вы только начали изучать базы данных, возможно, этот материал не принесёт вам много пользы, но если у вас есть хотя бы 2-3 года опыта работы с базами данных, настоятельно рекомендуется ознакомиться с этим фундаментальным трудом.

В завершении данной главы подчеркнём одну важную мысль, которую очень часто упускают из виду, — разницу между «отношением», «схемой отношения», «переменной отношения», «значением отношения».

Как легко догадаться, первый термин («отношение») повсеместно используют вместо трёх других (хотя он является синонимом термина «значение отношения»). Ещё чаще говорят просто «таблица» (что вполне справедливо в контексте языка SQL).

Так в чём же разница? (И добавим ещё один термин.)

- Схема отношения — описание атрибутов отношения (указание их имён, типов данных и иных свойств).
- Переменная отношения — реально созданная в СУБД таблица.
- Отношение (значение отношения) — те данные, которые в данный момент хранятся в переменной отношения.
- Производная переменная отношения (представление{340}) — дополнительная форма извлечения данных из отношения.

Очевидно, что значений у переменной отношения много (изменение хотя бы одного бита данных уже даёт новое значение).

Поясним все термины графически (рисунок 2.1.b) с указанием последовательности реализации: на основе модели отношения строится схема отношения, с помощью которой можно сформировать SQL-запрос для создания переменной отношения (а затем и представления{340}), после чего переменную отношения можно наполнять данными, что даст нам значение отношения и значение представления.

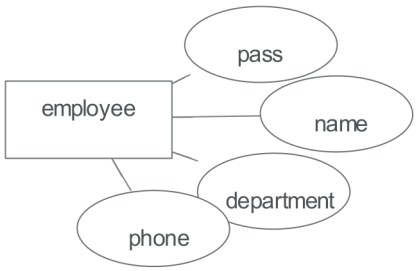
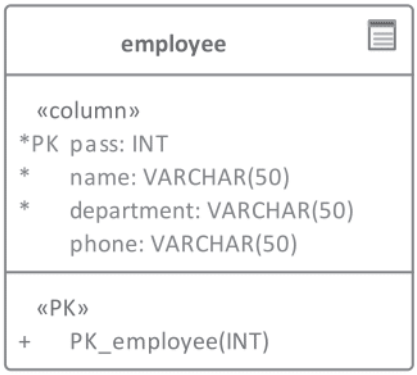
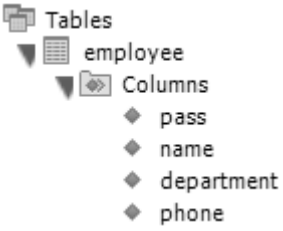
1) В процессе создания модели данных описываются сущности и их атрибуты:	
2) Модель детализируется до схемы отношения	
3) На основе имеющейся схемы отношения генерируется SQL-код для создания переменной отношения (т.е. реальной таблицы в реальной базе данных под управлением СУБД):	**MySQL** **Создание таблицы** `CREATE TABLE `employee` ( `pass` INT(11) NOT NULL, `name` VARCHAR(50) NOT NULL, `department` VARCHAR(50) NOT NULL, `phone` VARCHAR(50) DEFAULT NULL, PRIMARY KEY (`pass`) ) ENGINE=InnoDB DEFAULT CHARSET=utf8;`

*Рисунок 2.1.b — **Все термины, связанные с понятием «отношение»***

4) В СУБД появляется реальная таблица:



5) Теперь на основе таблицы можно построить представление, т.е. производную переменную отношения:

MySQL **Создание представления**

```
1   CREATE VIEW `pass_and_name`
2   AS
3    SELECT `pass`,
4           UPPER(`name`) AS `name`
5    FROM   `employee`
```

6) Пример значения отношения выглядит так:

pass	name	department	Phone
178	Иванов И.И.	Отд-1	*NULL*
223	Петров П.П.	Отд-1	999-87-32
318	Сидоров С.С.	Отд-1	333-55-66
243	Сидоров С.С.	Отд-2	333-55-66

7) Пример значения представления (производной переменной отношения) выглядит так:

pass	name
178	ИВАНОВ И.И.
223	ПЕТРОВ П.П.
318	СИДОРОВ С.С.
243	СИДОРОВ С.С.

*Рисунок 2.1.b (продолжение) — **Все термины, связанные с понятием «отношение»***

Чтобы не запутаться окончательно в терминологии, можно использовать следующую шпаргалку[26] (это разделение совершенно условно, но хорошо запоминается):

Точка зрения	Взаимное соответствие терминов		
	Группа 1	**Группа 2**	**Группа 3**
Предметной области	Сущность	Экземпляр	Свойство
Реляционной теории	Отношение	Кортеж	Атрибут
Физической базы данных	Таблица	Строка	Столбец
Физического хранения	Файл	Запись	Поле

На этом теоретическая часть закончена, и теперь мы рассмотрим, как дела обстоят в реальности.

Задание 2.1.a: приведите 15-20 примеров отношений, описывающих те или иные объекты реального мира. Особое внимание уделите типам данных, которые вы выберете для каждого атрибута.

Задание 2.1.b: каких отношений, на ваш взгляд, не хватает в базе данных «Банк»{408}? Дополните её модель недостающими отношениями.

Задание 2.1.c: каких атрибутов отношений, на ваш взгляд, не хватает в базе данных «Банк»{408}? Дополните соответствующие отношения пропущенными атрибутами.

[26] К сожалению, установить первоисточник этой таблицы так и не удалось, что нисколько не умаляет заслуг её автора и пользу от её применения.

2.1.2. СОЗДАНИЕ И ИСПОЛЬЗОВАНИЕ ОТНОШЕНИЙ

Сразу же начнём с не очень технологического, но критически важного замечания.

Всегда используйте для именования объектов базы данных **только английский** язык. Если вы его не знаете, пишите «транслитом», но всё равно используйте только английский алфавит.

Несмотря на то, что подавляющее большинство средств проектирования баз данных и СУБД поддерживают возможность именования объектов на русском языке (и многих других), всегда может возникнуть ситуация, в которой вся система перестанет работать из-за проблем с кодировками в одном из множества программных средств.

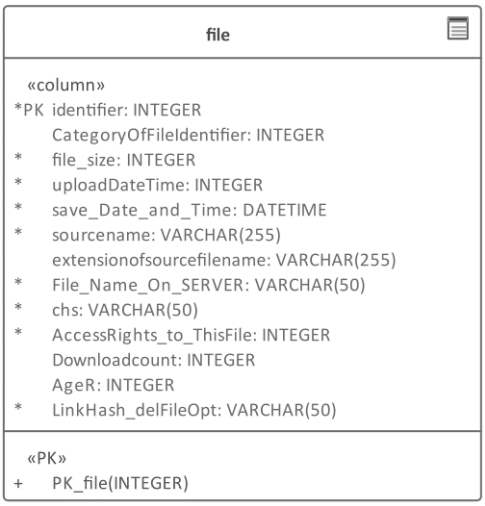
К тому же использование английского языка в IT-сфере — незыблемая общепринятая международная традиция.

В данной главе мы осознанно не будем рассматривать моделирование отношений на инфологическом уровне (этому вопросу посвящён соответствующий раздел{284}), а поговорим о технической реализации даталогического и физического уровней.

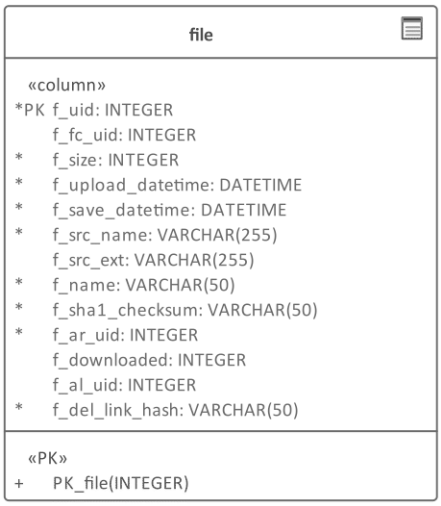
Конкретные действия на этих уровнях будут зависеть от возможностей используемого вами средства проектирования, но непосредственно в Sparx Enterprise Architect (где в контексте моделирования баз данных возможности даталогического уровня почти не отличаются от возможностей физического) на даталогическом уровне стоит сделать следующее.

Выработать стандарт именования объектов базы данных. Чтобы избежать бесполезных споров о том, какой стандарт лучше, подчеркнём лишь главную мысль: обязательно стоит следовать какому-то (пусть и созданному лично вами) одному стандарту. И только ему.

Сравните две схемы отношения, описывающего файл в некоем файлообменном сервисе (рисунок 2.1.c):



Плохой вариант



Хороший вариант

*Рисунок 2.1.c — **Пример схем отношения без соблюдения и с соблюдением стандарта именования***

В масштабах модели всей базы данных проблема несоблюдения стандартов именования превращается в катастрофу: в такой «мешанине» очень тяжело ориентироваться, очень легко пропустить ошибку, да и у профессионалов, привыкших к соблюдению стандартов, такое зрелище вызывает почти физическую боль.

Создать отношения и наполнить их атрибутами. Все необходимые отношения и все (по крайней мере, очевидные, «информационные») атрибуты к этому моменту уже известны (получены на предыдущем, инфологическом уровне моделирования). Так что смело можно переносить эту информацию в даталогическую модель. Пример того, как будет выглядеть получившееся отношение, можно увидеть на рисунке 2.1.c.

Указать самые очевидные ограничения. На этом уровне проектирования можно позволить себе не погружаться в детали технической реализации таблиц базы данных и учесть лишь те особенности, которые явным образом следуют из предметной области (например, требования уникальности значений некоторых атрибутов или требования наличия явного значения атрибута (свойство `NOT NULL`). На рисунке 2.1.b у атрибутов, слева от имени которых стоит знак «*», включено свойство `NOT NULL`.

Указать самые очевидные ключи{35}**, связи**{57}**, индексы**{106}**.** Здесь продолжается та же логика, что была указана в предыдущем пункте: отмечаем лишь самое важное — то, без чего модель данных явно начнёт терять адекватность предметной области{12}. Так, например, на рисунке 2.1.b у атрибута `f_uid` включено свойство `PK`, т.е. он является первичным ключом отношения.

С даталогическим уровнем — всё. Сложность здесь будет заключаться не в технической реализации (это действительно рутинная операция, почти не требующая умственных усилий), а в том, чтобы полноценно отразить модель данных предметной области на модель базы данных, при этом соблюдая все необходимые требования{12}.

На физическом же уровне ситуация усложняется. Во-первых, мы по-прежнему обязаны помнить о предметной области. Во-вторых, мы должны очень хорошо понимать внутреннюю логику работы выбранной СУБД, т.к. здесь мы будем принимать множество технических решений.

Очень часто можно услышать вопрос о том, существует ли возможность автоматической конвертации даталогической модели в физическую. К огромному сожалению — да, существует. И воспользоваться этой возможностью — значит совершить ошибку.

Даталогическая модель близка к инфологической и в первую очередь отвечает требованию адекватности предметной области{12}. Физическая же модель (не нарушая требований адекватности) должна в том числе обеспечивать удобство использования{14}, производительность{16}, защищённость данных{17}.

Очевидно, что автоматическая конвертация создаст вам «техническую копию» даталогической модели, не более. И только вдумчивое «ручное» создание физической модели позволяет получить по-настоящему качественную базу данных.

Итак, что необходимо сделать с отношениями на физическом уровне.

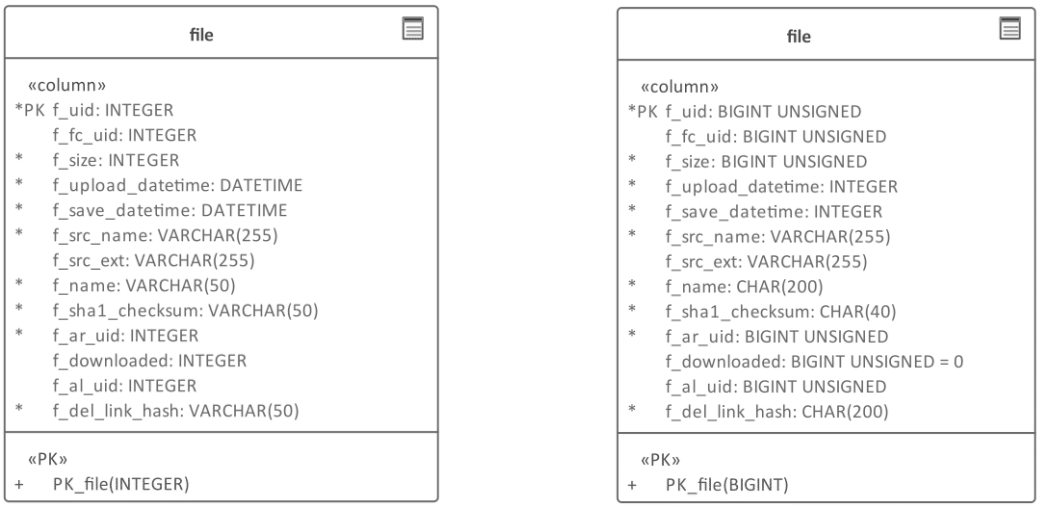
Конкретизировать типы данных. Здесь уже недостаточно в общих чертах понимать, что «вот тут — строка, а вот тут — число». Строки, числа, датавременные и денежные величины (и многие другие) выражаются вполне конкретными (и при том несколькими разными!) типами данных, поддерживаемыми конкретной СУБД.

Тщательное продумывание типов данных важно потому, что они влияют как на производительность (меняется объём обрабатываемой информации и количество дополнительных операций, выполняемых СУБД при анализе и конвертации данных), так и на адекватность предметной области (существует риск использовать такой тип данных, который не позволит сохранить какие-то объективно существующие в жизни значения или будет сохранять их с искажениями).

Например, если мы выберем слишком «большой» целочисленный тип (допустим, `BIGINT`) для хранения значений первичного ключа, мы снизим производительность. Но если мы выберем слишком «маленький» тип (допустим, `TINYINT`), мы сможем сохранить не более 256 различных значений (т.е. в таблицу можно будет поместить не более 256 записей).

Ещё одной прекрасной иллюстрацией опасности при выборе типов данных является тип `DATETIME` в MS SQL Server, который хранит дробное значение секунд с округлением до одного из трёх значений — .000, .003, .007. Т.е. вы никак не сможете сохранить здесь значения тысячных долей секунды, равные .001, .002, .004, .005, .006, .008, .009.

Примерная картина того, как меняются типы данных при переходе от даталогического уровня к физическому, показана на рисунке 2.1.d.



Даталогический уровень Физический уровень

*Рисунок 2.1.d — **Типы данных на даталогическом и физическом уровнях***

Указать все ограничения. И здесь речь идёт не только об уникальности значений или `NOT NULL`, здесь нужно создать **всё**: первичные и внешние ключи{35} (и доработать все необходимые связи{57}), уникальные индексы{109}, проверки{347}, триггеры{350}.

Указать доступные средства повышения производительности. К таким средствам в общем случае относятся индексы{106} (не путать с уникальными индексами{109}, которые создаются не столько для производительности, сколько для обеспечения уникальности значений), методы доступа{33} и их параметры, материализованные представления{340} и т.д.

Указать специфические свойства отношений и их атрибутов. К таким свойствам относятся настройки кодировок строковых типов данных, значения атрибутов по умолчанию, начальные значения и шаг инкремента автоинкрементируемых величин и т.д. (этот список очень сильно зависит от выбранной СУБД).

По количеству ссылок на другие разделы книги вы уже могли понять, что крайне сложно пытаться реализовать «отношения в чистом виде», в отрыве от других объектов базы данных. Потому многие идеи, лишь обозначенные здесь тезисно, будут подробно рассмотрены далее.

Сейчас же стоит упомянуть простой, но порождающий множество вопросов и ошибок момент — отличие «отношений» (как понятия реляционной теории) и «таблиц» (как повседневной реальности работы с СУБД).

См. очень подробное изложение данного вопроса в главе 6 книги «Введение в системы баз данных» (К. Дж. Дейт, 8-е издание).

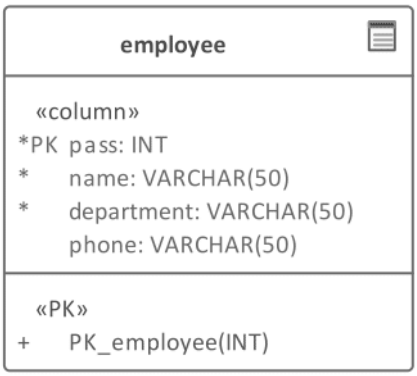
Начнём с того общего, что есть у «отношений» и «таблиц».

Сходство 1. У отношений есть именованные типизированные атрибуты, их аналогом в таблицах являются столбцы со своими именами и типами.

Сходство 2. У отношений есть набор кортежей, их аналогом в таблицах являются строки.

На этом сходства заканчивается, и начинаются отличия.

Отличие 1. Атрибуты отношения не упорядочены, в таблицах же последовательность столбцов задаётся при создании таблицы, но может быть изменена в рамках выполнения SQL-запросов и может не совпадать с реальным образом хранения данных. Допустим, у нас есть отношение, представленное следующей схемой и созданное на уровне СУБД следующим SQL-кодом (рисунок 2.1.е).



MySQL **Создание таблицы**

```
1  CREATE TABLE `employee`
2  (
3    `e_pass` INT NOT NULL,
4    `e_name` VARCHAR(100) NOT NULL,
5    `e_department` VARCHAR(10) NOT NULL,
6    `e_phone` VARCHAR(50) NULL,
7    CONSTRAINT `PK_employee`
8    PRIMARY KEY (`e_pass`)
9  )
```

*Рисунок 2.1.е — **Схема отношения и SQL-код для создания таблицы***

Выполняя запрос, мы можем написать так:

MySQL **Запрос, в котором порядок полей совпадает с реальным порядком полей в таблице**

```
1  SELECT `e_pass`,
2         `e_name`
3  FROM   `employee`
```

Но можем написать и вот так (поменяв произвольным образом порядок следования извлекаемых полей):

MySQL **Запрос, в котором порядок полей не совпадает с реальным порядком полей в таблице**

```
SELECT `e_name`,
       `e_pass`
FROM   `employee`
```

Что касается физического расположения данных на диске, то здесь вообще нет единого правила — СУБД (а точнее, т.н. метод доступа{33}) может в целях оптимизации размещать данные, относящиеся к каждому столбцу, совершенно неожиданным для человека образом.

Отличие 2. Кортежи отношения не упорядочены, в таблицах же строки хранятся в некоторой последовательности (как правило — в порядке, определяемом кластерным индексом{110}, если он есть у таблицы).

С этим отличием связан очень частый (и в корне неверный) вопрос: «Как мне вставить в таблицу новую строку между строками такой-то и такой-то?» Никак.

Тот факт, что СУБД вынуждена хранить данные в некоторой последовательности (это не только объективная необходимость, но и способ оптимизации операций поиска) никак не означает, что вам будет что-то известно об этой последовательности, и уж тем более, что СУБД позволит вам ею управлять. Т.е. если у нас есть строки

e_name
Иванов И.И.
Петров П.П.
Сидоров С.С.
Сидоров С.С.

и мы хотим поместить строку «Орлов О.О.» между строками «Петров П.П.» и «Сидоров С.С.», мы никак не сможем это сделать. Да и зачем?! Если нам нужно упорядочивание (например), по алфавиту, достаточно написать запрос с **`ORDER BY`**:

MySQL **Запрос, в котором указано правило упорядочивания результата по уже имеющемуся в таблице полю**

```
SELECT   `e_name`
FROM     `employee`
ORDER BY `e_name`
```

Если же нам нужно какое-то совершенно искусственное упорядочивание, не основанное на имеющихся в таблице данных, придётся добавить новое поле, и затем выполнять упорядочивание по его значению, например, так:

e_name	e_order
Иванов И.И.	10
Петров П.П.	20
Сидоров С.С.	40
Сидоров С.С.	50
Орлов О.О.	30

MySQL **Запрос, в котором указано правило упорядочивания результата по специально добавленному полю**

```
SELECT   `e_name`
FROM     `employee`
ORDER BY `e_order`
```

Результат будет таким (строка «Орлов О.О.» окажется между строками «Петров П.П.» и «Сидоров С.С.», как и требовалось, но физическое расположение данных на диске при этом не поменяется, т.е. этот эффект будет актуален только в рамках выполнения отдельного SQL-запроса):

e_name
Иванов И.И.
Петров П.П.
Орлов О.О.
Сидоров С.С.
Сидоров С.С.

Отличие 3. В отношениях не допускается наличие одноимённых атрибутов и совпадающих кортежей, в таблицах же (как минимум — производных, т.е. полученных выполнением запроса) это требование может нарушаться.

Начнём с дублирования строк. Почти любая современная реляционная СУБД позволит вам создать таблицу без первичных ключей{39} и/или уникальных индексов{109}, и затем размещать там полностью идентичные строки. Зачем вам это нужно, и как вы теперь собираетесь различать эти строки — отдельный вопрос (подсказка: не делайте так никогда, это приводит к уйме проблем), но технологически здесь нет никаких препятствий.

Что до одноимённых атрибутов, то СУБД не допустит такого при создании таблицы, но при выполнении SQL-запросов с `JOIN` такая ситуация возможна. Рассмотрим соответствующий пример.

На рисунке 2.1.f представлены схемы двух отношений, в которых сознательно созданы одинаковые наборы полей:

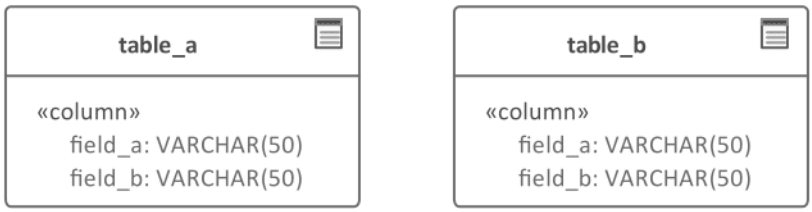


*Рисунок 2.1.f — **Схемы отношений для демонстрации эффекта дублирования имён атрибутов при выполнении SQL-запроса с `JOIN`***

Создадим соответствующие таблицы и наполним их данными (рисунок 2.1.g):

table_a

field_a	field_b
One	1
Two	2

table_b

field_a	field_b
One	1000
Two	2000

*Рисунок 2.1.g — **Данные в таблицах для демонстрации эффекта дублирования имён атрибутов при выполнении SQL-запроса с `JOIN`***

Теперь выполним следующий запрос (код приведён в синтаксисе MySQL, но отметим, что MS SQL Server ведёт себя аналогичным образом, а вот Oracle автоматически добавляет в конец дублирующегося имени постфикс «_1»):

MySQL **Запрос, результат выполнения которого будет содержать одноимённые поля (столбцы)**

```
1   SELECT *
2   FROM   `table_a`
3   JOIN   `table_b`
4   USING (`field_a`)
```

Результат выполнения запроса таков (обратите внимание, что здесь есть два столбца **field_b** с одинаковыми именами, но разными данными):

field_a	field_b	field_b
One	1	1000
Two	2	2000

Со сходствами и различиями отношений и таблиц на этом всё. И остаётся упомянуть ещё один момент, который традиционно обходят вниманием в книгах по базам данных, т.к. эта тема скорее уместна в документации по СУБД. Поговорим о том, что такое «метод доступа» (storage engine), и как он связан с отношениями и таблицами.

Метод доступа (storage engine, database engine[27]) — программный компонент, используемый СУБД для добавления, чтения, обновления и удаления данных.

Упрощённо: специальная часть СУБД, с помощью которой она работает с диском (файлами).

К сожалению, почти бессмысленно описывать «методы доступа в принципе», т.к. их набор, конкретная реализация, структура, возможности и т.д. весьма различны и жёстко связаны с конкретной СУБД. При этом многие СУБД позволяют использовать несколько методов доступа одновременно, что расширяет возможности проектирования базы данных на физическом уровне, но и усложняет его.

От выбора метода доступа зависит производительность различных операций, доступность (и особенность реализации) механизма транзакций{374}, необходимый для хранения данных объём дискового пространства, требования к процессору и оперативной памяти и т.д. и т.п.

Если это кажется слишком сложным (а так оно и есть), можно успокоить себя мыслью о том, что любая современная СУБД по умолчанию использует «наиболее сбалансированный» метод доступа, ориентированный в первую очередь на надёжность операций с данными (как правило, ценой потери производительности и повышения требований к аппаратным ресурсам). Если же вы всерьёз хотите оптимизировать свою базу данных на физическом уровне, придётся изучить соответствующий раздел документации той СУБД, с которой вы работаете.

В завершении данной главы ещё раз подчеркнём, что для успешного создания модели базы данных требуется хорошее понимание принципов работы и возможностей как выбранного средства проектирования, так и целевой СУБД. Потому тщательно изучайте соответствующую документацию — это сэкономит вам уйму времени и позволит избежать множества ошибок.

А продолжение логики создания и использования отношений будет в разделах, посвящённых практической реализации ключей{50}, связей{78} и триггеров{350}.

---

27 **Database engine (storage engine)** — the underlying software component that a database management system (DBMS) uses to create, read, update and delete (CRUD) data from a database. («Wikipedia») [https://en.wikipedia.org/wiki/Database_engine]

Задание 2.1.d: с помощью любого средства проектирования создайте модели 3-5 отношений из тех, что вы придумали, выполняя задание 2.1.a. После создания моделей сгенерируйте соответствующий SQL-код и выполните его в выбранной вами СУБД, чтобы создать таблицы. Наполните получившиеся таблицы примерами данных.

Задание 2.1.e: все ли отношения и их атрибуты в базе данных «Банк»{408} следуют единому стандарту именования объектов базы данных? Предложите свои улучшения.

Задание 2.1.f: для всех ли атрибутов отношения в базе данных «Банк»{408} выбраны оптимальные типы данных? Предложите свои улучшения.

2.2. КЛЮЧИ

2.2.1. ОБЩИЕ СВЕДЕНИЯ О КЛЮЧАХ

Поскольку каждая разновидность ключей обладает своими особыми свойствами, определение ключа как такового можно сформулировать лишь в общем виде:

Ключ (key[25]) — идентификатор, являющийся частью набора элементов данных. В контексте реляционных СУБД ключом считается совокупность атрибутов отношения (или отдельный атрибут), обладающий определёнными, характерными для данного вида ключа свойствами.

Упрощённо: поле (или набор полей), подчиняющееся определённым правилам (конкретные правила зависят от вида ключа).

На рисунке 2.2.а представлен общий перечень ключей и их взаимосвязь друг с другом.

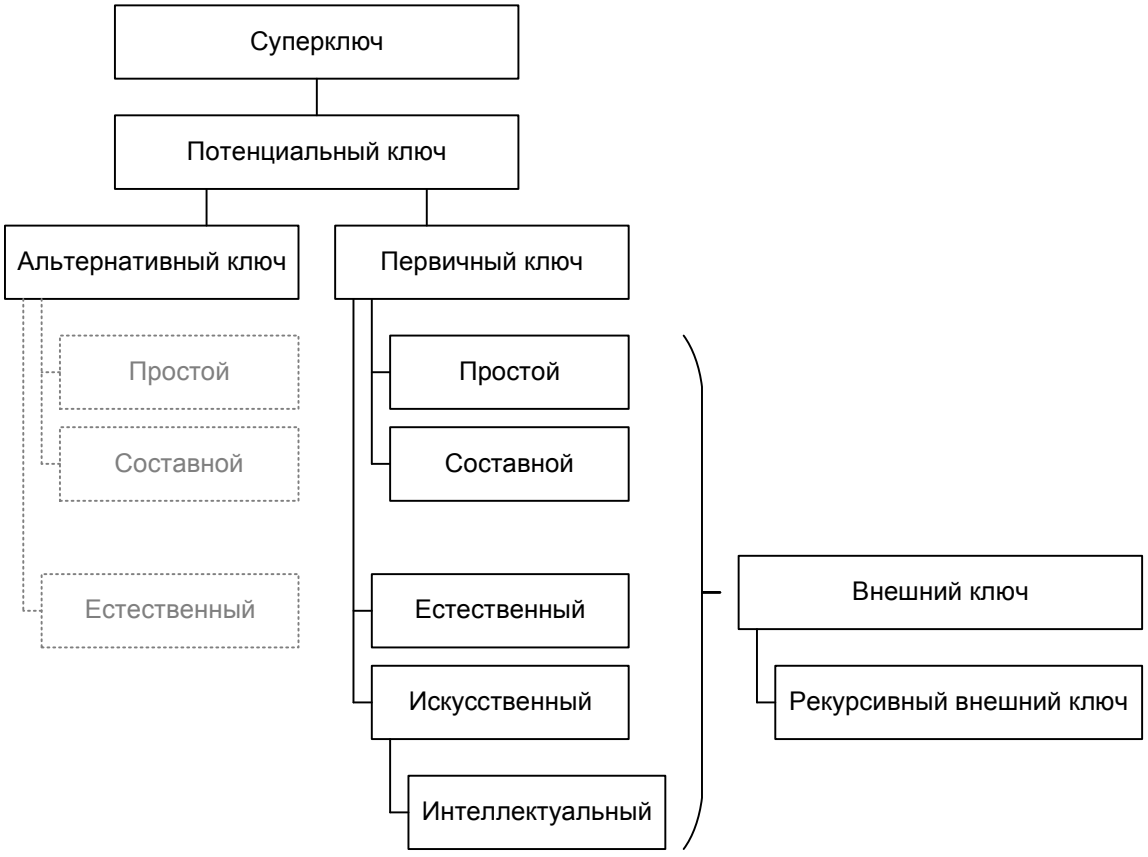


*Рисунок 2.2.а — **Виды ключей и их взаимосвязь***

Сразу же отметим, что искусственные (и интеллектуальные) альтернативные ключи теоретически могут существовать, но практически не имеют никакого смысла (более того — скорее вредят, повышая нагрузку на СУБД).

[28] **Key** — identifier that is part of a set of data elements (ISO/IEC 2382:2015, Information technology — Vocabulary).

Итак, теперь последовательно рассмотрим все определения и приведём соответствующие примеры.

Суперключ (superkey[29]) — подмножество атрибутов отношения, уникально идентифицирующее любой кортеж. Суперключ часто называют надмножеством потенциального ключа, т.к. он может содержать в себе «лишние» элементы, удаление которых не приведёт к потере уникальности значений (т.к. суперключ не обладает свойством несократимости).

Упрощённо: поле или набор полей, по совокупности значений которых одну строку таблицы можно гарантированно отличить от другой; в таком наборе могут оказаться лишние поля, убрав которые, мы всё равно сможем гарантированно отличать одну строку от другой.

Несмотря на то, что на практике нас редко будут интересовать суперключи (они нужны, как правило, лишь в контексте нормализации{161}), поясним идею на примере уже многократно рассмотренного отношения `employee` (рисунок 2.2.b).

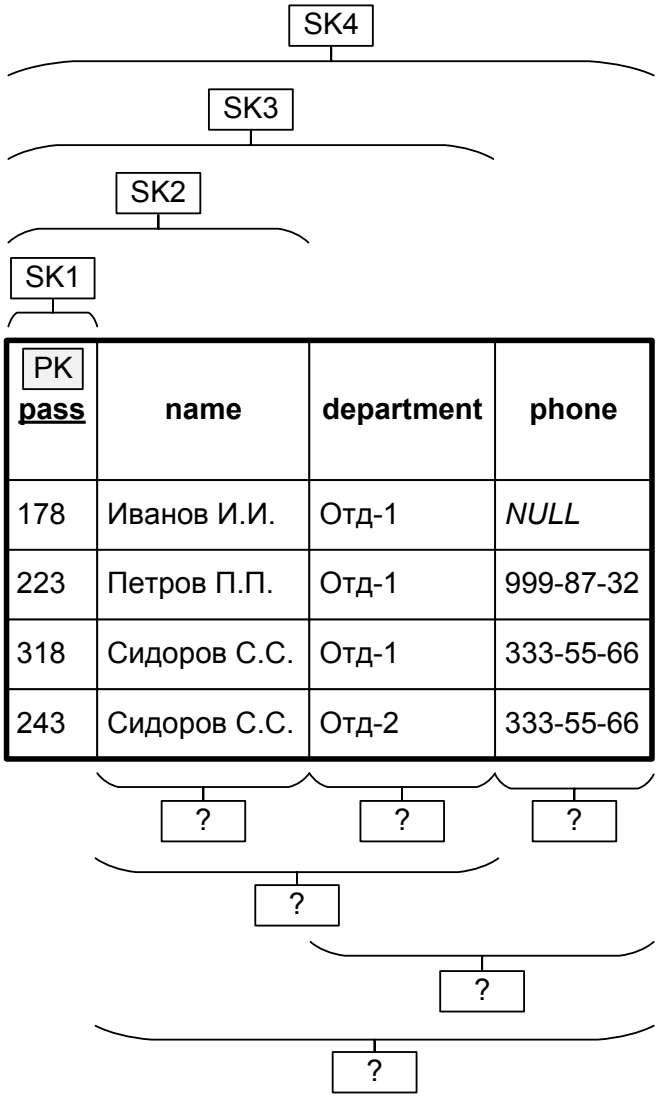


*Рисунок 2.2.b — **Суперключи отношения** `employee`*

[29] **Superkey** — a superset of a candidate key. A superkey has the uniqueness property but not necessarily the irreducibility property. Of course, a candidate key is a special case of a superkey. (C.J. Date, "An Introduction to Database Systems", 8th edition).

Обратите внимание, что все множества («SK1», «SK2», «SK3», «SK4»), включающие в себя первичный ключ (атрибут **pass**), объективно являются суперключами (уникально идентифицируют любой кортеж), в то время как множества, отмеченные «?», суперключами не являются: эти атрибуты или такие их совокупности могут содержать неуникальные значения.

Потенциальный ключ (candidate key[30]) — несократимое подмножество атрибутов отношения, уникально идентифицирующее любой кортеж. Иными словами, потенциальный ключ — это такой суперключ, в котором нет «лишних» элементов. Как правило, один из потенциальных ключей в дальнейшем становится первичным ключом (а остальные потенциальные ключи становятся альтернативными ключами).

Упрощённо: кандидат в первичные ключи, т.е. поле или набор полей, по совокупности значений которых одну строку таблицы можно гарантированно отличить от другой; в таком наборе нет лишних полей, т.е. убрав из набора хотя бы одно поле, мы уже не сможем гарантированно различить любые строки таблицы.

В рассмотренном на рисунке 2.2.b примере потенциальным является только суперключ «SK1», т.к. только атрибут **pass** содержит гарантированно уникальные значения для каждого кортежа. Остальные комбинации атрибутов или не гарантируют уникальности (такие комбинации на рисунке 2.2.b отмечены «?»), или включают в себя лишние элементы, удаление которых не приведёт к потере уникальности («SK2», «SK3», «SK4»).

В общем случае в отношении может быть несколько потенциальных ключей, что очень легко продемонстрировать на примере отношения **car** (рисунок 2.2.c).

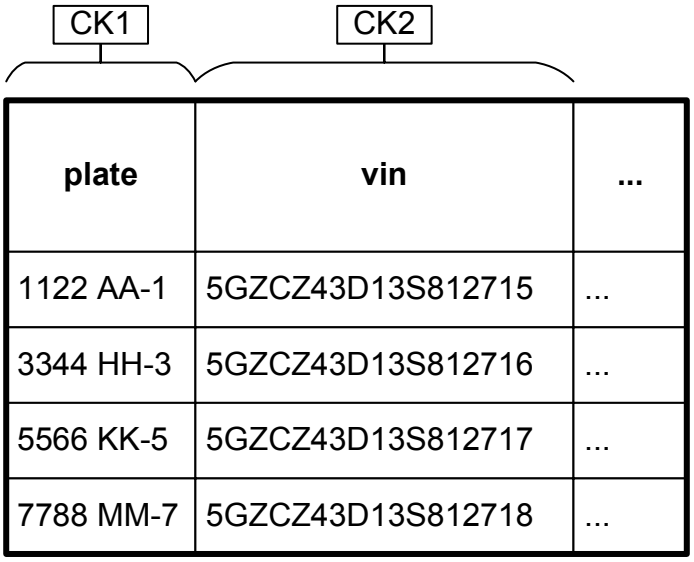


*Рисунок 2.2.c — **Пример отношения с несколькими потенциальными ключами***

Поскольку у машины уникальны как номер госрегистрации (атрибут **plate**), так и VIN-номер (атрибут **vin**), значение любого из этих атрибутов гарантированно идентифицирует любой кортеж отношения, т.е. каждый из этих атрибутов является потенциальным ключом.

[30] **Candidate key** — a set of attributes that has both of the following properties: uniqueness and irreducibility. (C.J. Date, "An Introduction to Database Systems", 8th edition).

Как и было только что отмечено в определении потенциального ключа, как правило, один из таких ключей становится первичным ключом, а остальные становятся альтернативными ключами.

Альтернативный ключ (alternate key[31]) — потенциальный ключ отношения, не выбранный в качестве первичного ключа.

Упрощённо: кандидат в первичные ключи, так и не ставший первичным ключом.

Если продолжить рассмотрение представленного на рисунке 2.2.c отношения **`car`**, то после выбора атрибута **`plate`** в качестве первичного ключа (скоро будет объяснено, почему **`vin`** хуже подходит на эту роль{39}), оставшийся потенциальный ключ в виде атрибута **`vin`** становится альтернативным ключом.

Альтернативные ключи требуют особого внимания. По определению их значения гарантированно идентифицируют кортежи отношения (т.е. не могут дублироваться), но СУБД (пока) ничего об этом «не знает» и контролирует уникальность только значений первичного ключа.

Чтобы привести поведение СУБД в соответствие с требованиями предметной области, на альтернативных ключах создают т.н. «ограничение уникальности» (**`CHECK UNIQUE`**), которое в большинстве СУБД обозначает построение на данном ключе т.н. «уникального индекса{109}».

Изобразим всё только что описанное на рисунке (см. рисунок 2.2.d).

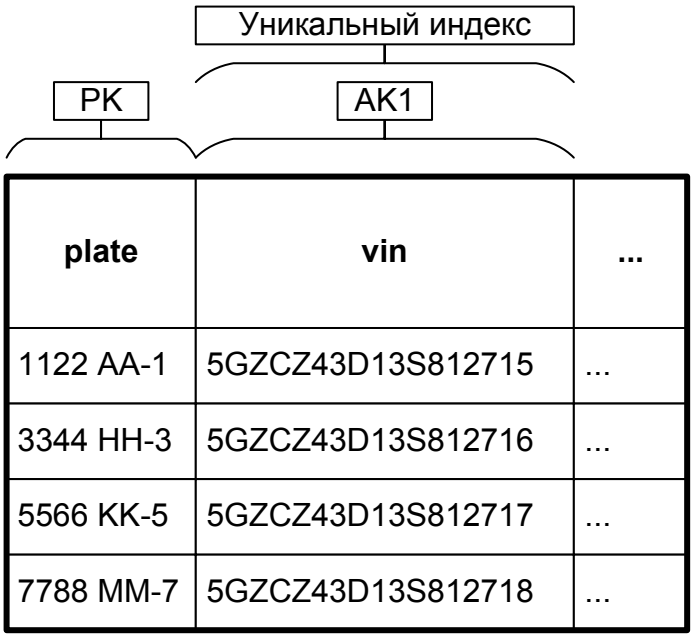


*Рисунок 2.2.d — **Первичный и альтернативный ключи***

Обратите внимание, что в имени альтернативного ключа «AK1» присутствует индекс («1»), т.к. у отношения теоретически может быть много альтернативных ключей, в то время как первичный ключ («PK») у отношения всегда один.

Разновидности альтернативных ключей («простой», «составной», «естественный») совпадают с аналогичными разновидностями первичных ключей (и будут рассмотрены далее).

---

[31] **Alternate key**. It is possible for a given relvar to have two or more candidate keys. In such a case the relational model requires that exactly one of those keys be chosen as the primary key, and the others are than called **alternate keys**. (C.J. Date, "An Introduction to Database Systems", 8th edition).

Также отметим, что «искусственный» (и его подвид — «интеллектуальный») альтернативный ключ хоть и может существовать в теории, на практике он не имеет смысла, т.к. не даёт никаких преимуществ по управлению данными, а только лишь создаёт дополнительную «паразитную» нагрузку на СУБД.

Первичный ключ (primary key[32], PK) — потенциальный ключ, выбранный в качестве основного средства гарантированной идентификации кортежей отношения.

Упрощённо: уникальный идентификатор строки в таблице.

Из ранее рассмотренных определений следует, что первичный ключ должен обладать свойством несократимости (т.е. не должен включать в себя «лишние» элементы, которые можно отбросить без потери уникальности значений).

Также (поскольку понятие «первичного ключа» относится уже не столько к общей теории баз данных, сколько к практической работе СУБД) стоит отметить, что первичный ключ должен обладать свойством минимальности, которое следует трактовать буквально[33]: если существует несколько потенциальных ключей, состоящих из одинакового количества атрибутов, стоит выбрать в качестве первичного тот потенциальный ключ, который «физически меньше», т.е. содержит меньше данных.

Именно поэтому в ранее рассмотренном примере{38} (рисунок 2.2.d) в качестве первичного ключа был выбран атрибут `plate`, а не `vin`: чем меньше данных СУБД придётся сравнивать (для обеспечения свойства уникальности значений), тем быстрее она выполнит эту операцию.

Итак, самые основные факты о первичном ключе, которые важно понимать на данный момент:

- значения первичного ключа не могут дублироваться (т.е. они уникальны для каждой строки таблицы);
- первичный ключ должен быть несократимым (т.е. не содержать «лишних» элементов) и минимальным (т.е. в качестве первичного ключа должен быть выбран самый «короткий» потенциальный ключ);
- первичный ключ используется для гарантированной однозначной идентификации строки таблицы (что следует из свойства уникальности его значений) и для организации связей (будет рассмотрено далее, см. «внешний ключ{47}» и «связи{57}»);
- первичный ключ не может содержать неопределённое значение (`NULL`) — это свойство следует из логики обеспечения ссылочной целостности{72}.

Дальнейшие рассуждения о первичных ключах мы будем проводить в контексте их разновидностей. И первая из классификаций первичных ключей делит их на простые и составные.

---

32 **Primary key** — key that identifies one record (ISO/IEC 2382:2015, Information technology — Vocabulary).

33 В реляционной теории понятия «несократимость» и «минимальность» часто используют как синонимы, но здесь мы осознанно разделяем их, чтобы подчеркнуть два, по сути, разных свойства первичного ключа.

Простой ключ (simple key[34]) — ключ, состоящий из одного атрибута отношения.

Упрощённо: ключ, построенный на ровно одной колонке таблицы.

Составной ключ (compound key / composite key[35]) — ключ, состоящий из двух и более атрибутов отношения.

Упрощённо: ключ, построенный на двух (и более) колонках таблицы.

Как вы могли заметить, в этих определениях отсутствует слово «первичный», т.к. оба эти определения в равной мере касаются и первичных, и альтернативных, и внешних ключей.

Некоторые авторы подчёркивают, что термин «compound key» относится строго к внешним ключам, в то время как «composite key» — к потенциальным (а затем к первичным и альтернативным). Именно для того, чтобы исключить эти терминологически споры, англоязычные определения в сносках взяты из книги 2015-го года издания «The New Relational Database Dictionary» за авторством К. Дж. Дейта, где подобного разделения нет, а термины «compound key» и «composite key» явно обозначены как синонимы.

В качестве примера простого и составного ключей рассмотрим уже знакомое нам отношение **`employee`**.

Ранее мы исходили из предположения, что номера пропусков уникальны (не повторяются в масштабах всей фирмы), и именно такая ситуация отражена на рисунке 2.2.е — атрибут pass является классическим случаем простого первичного ключа.

*Рисунок 2.2.е — **Простой первичный ключ***

[34] **Simple key** — a key that's not composite. («The New Relational Database Dictionary», C.J. Date)

[35] **Compound key / composite key** — terms used interchangeably to mean a key consisting of two or more attributes. Contrast simple key. («The New Relational Database Dictionary», C.J. Date)

Но если бы номера пропусков были уникальными только в рамках отделов, для того, чтобы гарантированно отличить одного сотрудника от другого, нам нужно было бы знать **и** номер пропуска (`pass`), **и** название отдела (`department`), что вынудило бы нас создать составной первичный ключ на атрибутах `pass` и `department`. Эта ситуация показана на рисунке 2.2.f.

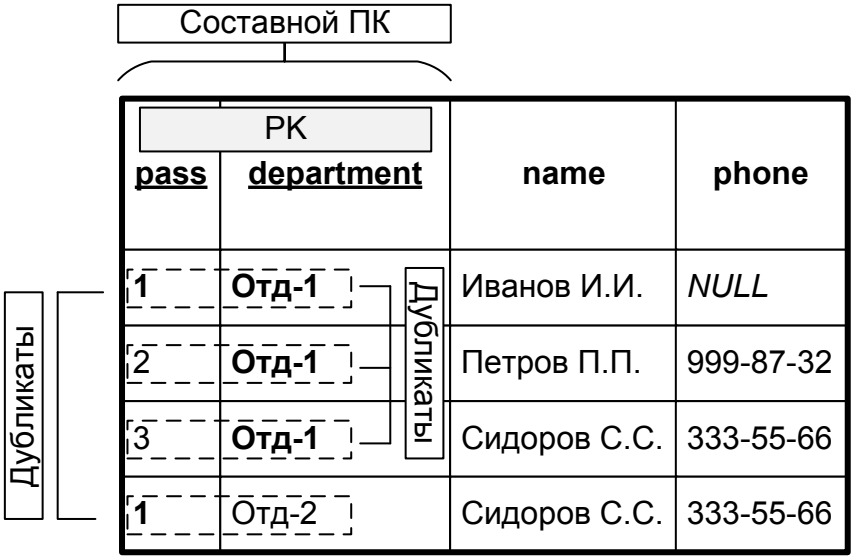


*Рисунок 2.2.f — **Составной первичный ключ***

Как следует из изображённого на рисунке 2.2.f, и атрибут `pass`, и атрибут `department` по отдельности допускают наличие дубликатов значений (т.е. не могут самостоятельно выступать в роли первичного ключа), но их совокупность отвечает требованию уникальности значений, и потому может быть первичным ключом.

С теоретической точки зрения (в силу того, что атрибуты отношения считаются неупорядоченными) безразлично, какой из атрибутов в составном первичном ключе будет идти первым (даже правильнее будет сказать, что в рамках реляционного подхода невозможно выяснить, какой атрибут идёт первым, вторым и т.д.), но на практике последовательность полей в составных ключах и индексах **критически важна** — об этом будет сказано уже совсем скоро{52}.

Теперь поговорим о естественных, искусственных и интеллектуальных ключах.

Для альтернативных ключей из этой классификации характерен только один вариант: альтернативные ключи в реальности бывают только естественными.

Искусственный и интеллектуальный ключи крайне редко «видны» на инфологическом уровне моделирования и «появляются» на даталогическом или даже физическом уровнях.

Справедливости ради стоит отметить, что интеллектуальные ключи в последние годы практически не используются.

Итак, рассмотрим определения.

Естественный ключ (natural key[36], business key) — ключ, построенный на множестве атрибутов отношения, несущих смысловую нагрузку.

Упрощённо: если у отношения есть атрибуты, значения которых в реальном мире уникальны (например, номер паспорта, номер госрегистрации автомобиля), построенный на них ключ и будет естественным.

Искусственный ключ, суррогатный ключ (surrogate key[37]) — ключ, построенный на атрибуте, искусственно добавленном в отношение с единственной целью — гарантированно идентифицировать кортежи отношения.

Упрощённо: в отношение добавляется новый атрибут, который никак не связан с предметной областью, а нужен исключительно для гарантированной идентификации отдельных записей.

Интеллектуальный ключ (intelligent key[38]) — ключ, значения которого не только идентифицируют кортежи отношения, но и несут дополнительную информацию.

Упрощённо: ключ, из значений которого можно извлечь некоторую информацию о кортеже, который он идентифицирует.

Графически естественный первичный ключ представлен на рисунке 2.2.g.

*Рисунок 2.2.g — **Естественный первичный ключ***

К преимуществам такого ключа можно отнести следующее:

- соответствующий атрибут уже есть в отношении, т.е. нет необходимости что-то добавлять (и хранить, расходуя лишнюю память);
- в случаях, когда это не противоречит требованиям безопасности, соответствующее значение может быть показано пользователям (и использовано ими для облегчения своей работы);

---

36 **Natural key** — term sometimes used to refer to a key that's not a surrogate key. («The New Relational Database Dictionary», C.J. Date)

37 **Surrogate key** — a single-attribute (i.e., simple) key with the property that its values serve solely as surrogates for the entities they're supposed to stand for. In other words, those surrogate key values serve merely to represent the fact that the corresponding entities exist, and they carry absolutely no additional information or meaning. («The New Relational Database Dictionary», C.J. Date)

38 **Intelligent key** — a single-attribute key whose values, in addition to their main purpose of serving as unique identifiers (typically for certain real world "entities"), carry some kind of encoded information embedded within themselves. («The New Relational Database Dictionary», C.J. Date)

- в большинстве СУБД и методов доступа{33} использование естественного первичного ключа позволяет получить очень полезный первичный индекс{110}, активно используемый в большинстве операций с таблицей.

Увы, недостатков у естественного ключа намного больше, и они очень серьёзные:

- если по соображениям безопасности значение атрибута, выбранного естественным первичным ключом, нельзя использовать в некоторых операциях, это делает соответствующие операции невозможными;
- размер естественного первичного ключа (за редким исключением) больше, чем размер искусственного первичного ключа (часто — в разы больше);
- если по каким-то причинам в реальной жизни мы не знаем значения естественного первичного ключа, мы или не сможем добавить запись в таблицу, или будем вынуждены «выдумать» значение (что ещё хуже);
- в некоторых отношениях естественный первичный ключ может быть только составным;
- если значение естественного первичного ключа изменится (например, у человека поменялся номер паспорта), СУБД будет вынуждена выполнить множество каскадных операции{73} для поддержания ссылочной целостности{72} (этот случай проиллюстрирован на рисунке 2.2.h: при изменении номера паспорта сотрудника СУБД будет вынуждена обновить все записи о платежах, чтобы по-прежнему «знать», к какому сотруднику они относятся).

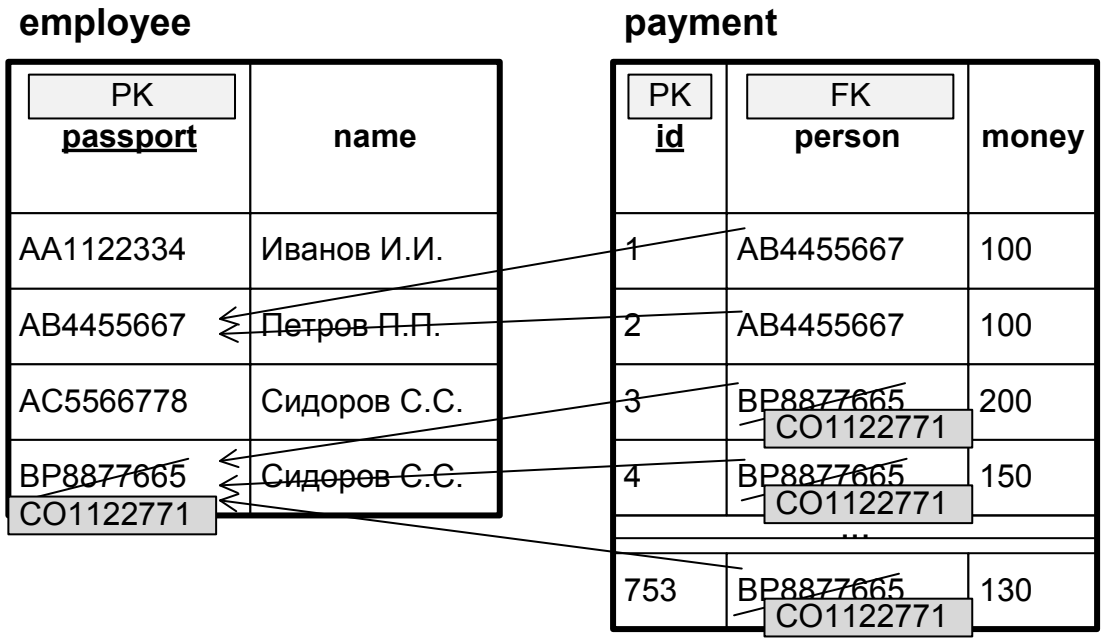


*Рисунок 2.2.h — **Каскадное обновление при изменении значения естественного первичного ключа***

Преимущества и недостатки естественного первичного ключа превращаются в, соответственно, недостатки и преимущества искусственного первичного ключа (показан на рисунке 2.2.i).

К недостаткам искусственного первичного ключа можно отнести следующее:

- соответствующий атрибут необходимо добавить в отношение (и хранить, расходуя лишнюю память);
- значение этого атрибута не имеет никакого «реального смысла», и потому даже если его показ пользователям не противоречит требованиям безопасности, они всё равно (скорее всего) не смогут использовать его для упрощения своей работы;
- в некоторых СУБД и методах доступа{33} нет возможности создать отдельный от первичного ключа первичный{110} индекс, и потому физическая организация таблицы и многие операции с ней будут реализовываться неэффективно.

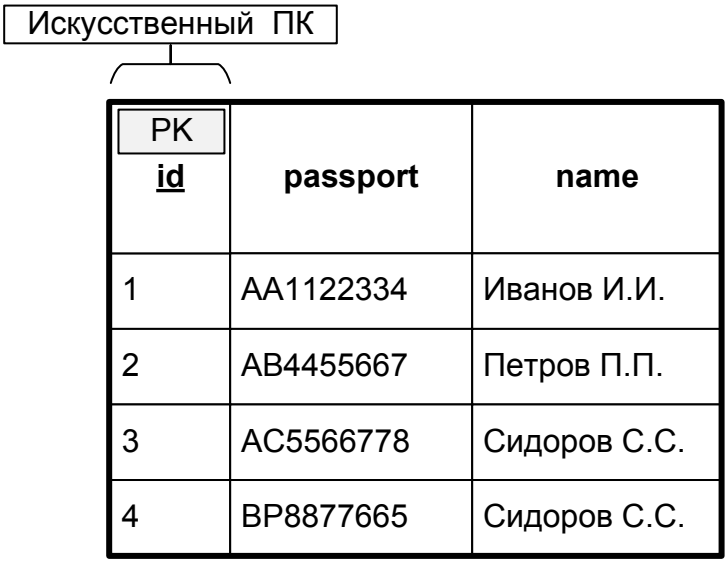


Рисунок 2.2.i — Искусственный первичный ключ

Преимущества искусственного первичного ключа:

- поскольку значение такого ключа не несёт никакого реального смысла (не содержит никаких «реальных данных»), как правило, на его использование налагается намного меньше ограничений безопасности;
- размер искусственного первичного ключа (за редким исключением) меньше, чем размер естественного первичного ключа (часто — в разы меньше);
- значение искусственного первичного ключа (как правило) генерируются автоматически, потому вероятность ситуации, когда при добавлении записи в таблицу мы «не будем знать» это значение, исчезающе мала;
- искусственный первичный ключ почти всегда будет простым;
- значение искусственного первичного ключа не меняется, а потому у СУБД нет необходимости выполнять множество каскадных операции{73} для поддержания ссылочной целостности{72} (этот случай проиллюстрирован на рисунке 2.2.j).

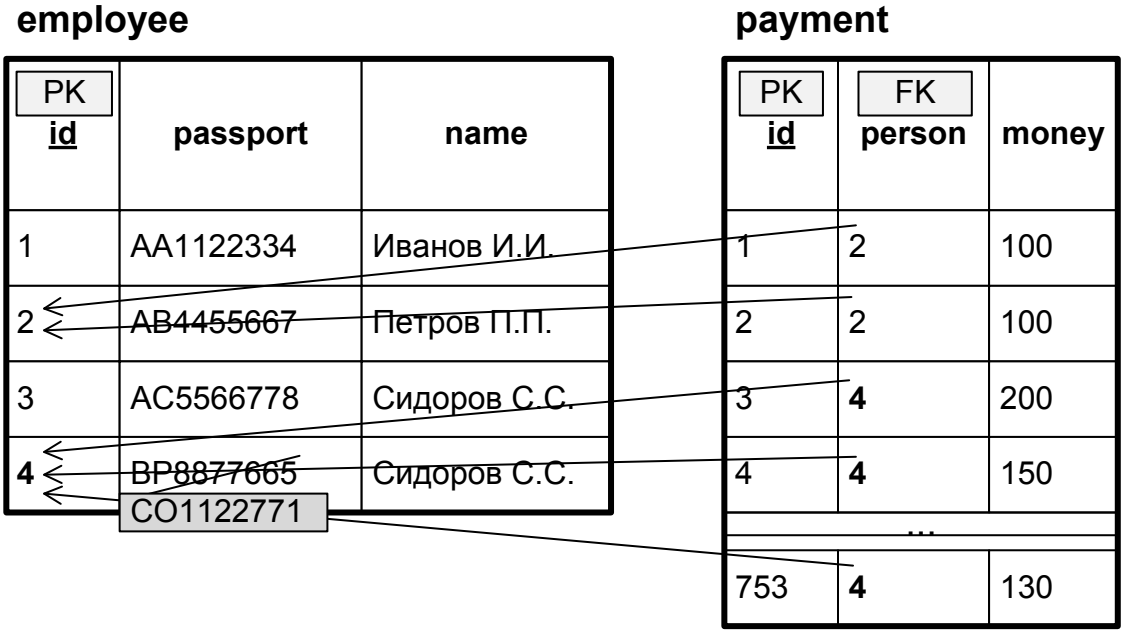


*Рисунок 2.2.j — **Отсутствие каскадного обновления при изменении значения искусственного первичного ключа***

В силу того факта, что преимуществ у искусственного первичного ключа очень много, и они крайне весомы, общепринятой практикой является создание таких ключей даже в тех отношениях, где имеется естественный потенциальный ключ.

При возможности использования естественного ключа, который не обладает перечисленными выше недостатками{43}, всё же стоит использовать его — это логично как минимум с точки зрения теории баз данных.

Почему же тогда во многих базах данных их разработчики «насильно» помещают в каждую таблицу (кроме промежуточных таблиц связей «многие ко многим{60}») суррогатный первичный ключ? Из-за удобства использования{14}: да, можно сколь угодно долго спорить о том, что такой подход неэффективен и нелогичен, но если он позволяет уменьшить количество ошибок в разрабатываемом продукте, его используют.

В любом случае, принимая решение в пользу суррогатного или естественного первичного ключа, стоит взвесить преимущества и недостатки обоих вариантов решения.

И ещё реже, чем естественные первичные ключи, встречаются интеллектуальные первичные ключи (пример такого ключа показан на рисунке 2.2.k).

Суть идеи интеллектуального первичного ключа состоит в том, чтобы по некоторому алгоритму генерировать относительно небольшой объём данных[39], не только гарантированно идентифицирующий любой кортеж отношения, но и содержащий в себе какую-то полезную информацию.

[39] Фактически, интеллектуальный ключ можно рассматривать как частично обратимую хэш-функцию (по хэш-значению можно восстановить часть исходных данных).

*Рисунок 2.2.k — **Интеллектуальный первичный ключ***

На рисунке 2.2.k такой ключ содержит в себе серию паспорта, первые две буквы фамилии и инициалы каждого человека, а числовой индекс в конце добавлен на случай т.н. «коллизий» (если все эти данные у двух и более людей совпадут, тогда индекс будет увеличиваться на единицу при каждом таком совпадении, чтобы итоговое значение ключа всё же было уникальным).

В теории предполагалось, что интеллектуальные ключи будут сочетать в себе преимущества естественных и искусственных ключей, при этом будучи избавленными от их недостатков. В реальности получилось строго наоборот — такие ключи почти всегда сочетают в себе недостатки естественных и искусственных, и не обладают их преимуществами. К тому же необходимо каждый раз реализовывать достаточно сложный (и медленный) механизм генерации таких ключей.

Несмотря на то, что в отдельных случаях интеллектуальные ключи даже могут оказаться полезными, общепринятая практика использования искусственных ключей полностью вытеснила интеллектуальные.

Итак, с первичными ключами — всё. Теперь мы переходим к рассмотрению совершенно иного вида ключей — внешних (и их подвида — рекурсивных внешних ключей).

Внешний ключ (foreign key[40], FK) — атрибут (или группа атрибутов) отношения, содержащий в себе копии значений первичного ключа другого отношения.

Упрощённо: поле в дочерней (подчинённой) таблице, содержащее в себе копии значений первичного ключа родительской (главной) таблицы.

Рекурсивный внешний ключ (recursive foreign key[41], RFK) — атрибут (или группа атрибутов) отношения, содержащий в себе копии значений первичного ключа этого же отношения.

Упрощённо: внешний ключ, находящийся в той же таблице, что и первичный ключ, значения которого он содержит; вариант внешнего ключа в случае, когда таблица ссылается на саму себя.

Внешние ключи мы уже видели на рисунках 2.2.h и 2.2.j (атрибут **`person`**), когда обсуждали естественные и искусственные ключи. Ещё раз эта идея проиллюстрирована на рисунке 2.2.l.

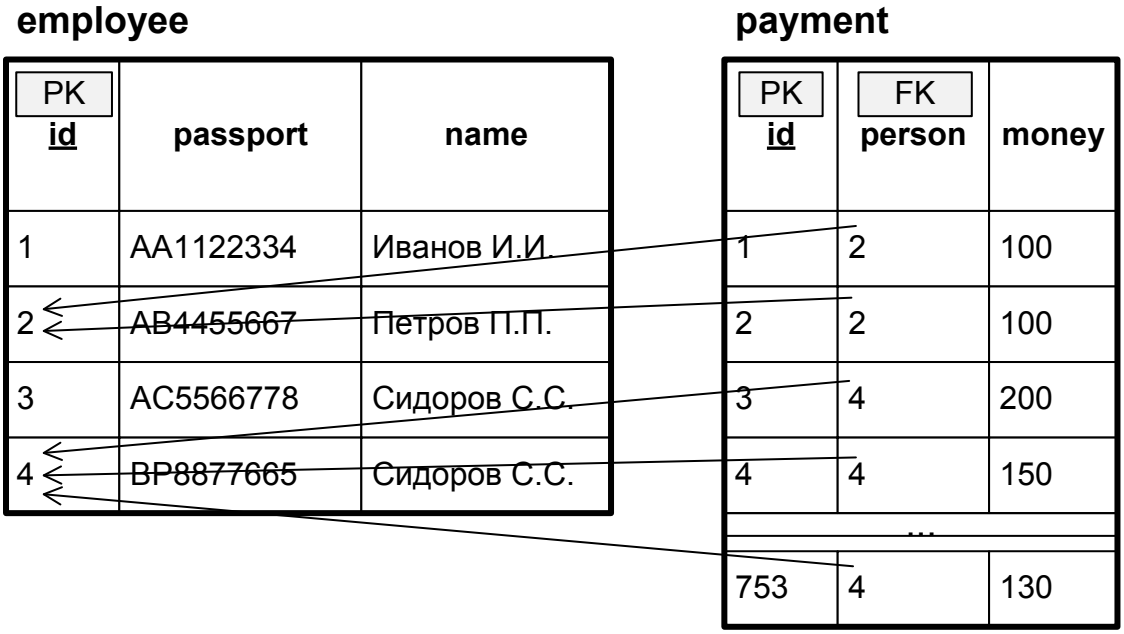


*Рисунок 2.2.l — **Внешний ключ***

Здесь таблица **`employee`** является родительской (главной), а таблица **`payment`** — дочерней (подчинённой). Соответственно, первичный ключ родительской таблицы (атрибут **`id`**) мигрировал в дочернюю таблицу и стал там внешним ключом (атрибут **`person`**). Уже на данном этапе очевидно, что имя внешнего ключа **не** обязано совпадать с именем первичного ключа, на которой он ссылается.

Продолжив рассмотрение рисунка 2.2.l, обратим внимание на стрелки, которыми показано, к какой записи родительской таблицы относится каждая из записей дочерней таблицы (платежи 1 и 2 относятся к сотруднику 2, а платежи 3, 4 и 753 относятся к сотруднику 4).

---

[40] **Foreign key** — in a relation, one or a group of attributes that corresponds to a primary key in another relation (ISO/IEC 2382:2015, Information technology — Vocabulary).

[41] **Recursive foreign key** — a foreign key that references some key of relation itself. («The New Relational Database Dictionary», C.J. Date)

Очевидно, что внешний ключ не обязан обладать свойством уникальности значений (легко заметить, что у атрибута **`person`** значения 2 и 4 повторяются несколько раз), хотя и может им обладать (в случае связи «один к одному[62]»).

Теперь рассмотрим, как работают рекурсивные внешние ключи. Классическим случаем их использование является организация иерархических структур, например, карты сайта, которая представлена на рисунке 2.2.m.

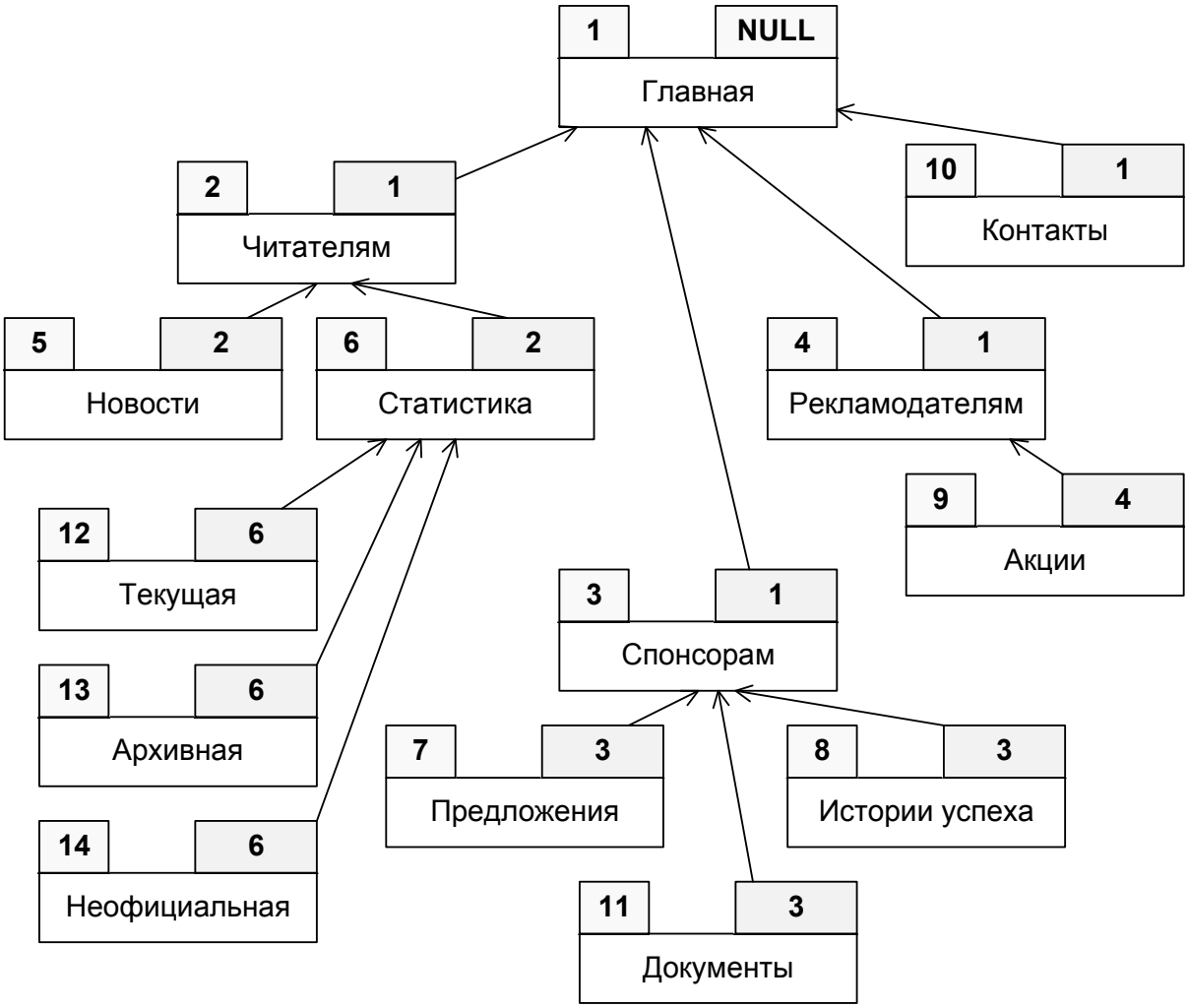


*Рисунок 2.2.m — **Визуальное представление карты сайта***

Технически для организации такой структуры в реляционной базе данных можно использовать отношение, представленное на рисунке 2.2.n.

Так у страницы «Главная» нет родительской страницы (значение внешнего ключа этой записи равно **`NULL`**), а для всех остальных записей в поле **`parent`** содержится значение первичного ключа соответствующей родительской записи.

Несмотря на то, что реляционные СУБД плохо приспособлены для организации подобных иерархических структур, представленное в данном примере решение является повсеместно распространённым и очень активно используется.

В книге[42] «Работа с MySQL, MS SQL Server и Oracle в примерах» приведено несколько задач (с решениями) как раз на обработку именно этой ситуации.

sitemap

PK **id**	RFK **parent**	**name**
1	NULL	Главная
2	1	Читателям
3	1	Спонсорам
4	1	Рекламодателям
5	2	Новости
6	2	Статистика
7	3	Предложения
8	3	Истории успеха
9	4	Акции
10	1	Контакты
11	3	Документы
12	6	Текущая
13	6	Архивная
14	6	Неофициальная

*Рисунок 2.2.n — **Организация карты сайта с использованием рекурсивного внешнего ключа***

Если обобщить описанное выше одной фразой, получается, что внешние ключи используются для организации связей между таблицами (или даже внутри одной таблицы), и дальнейшие рассуждения на эту тему будут продолжены в соответствующей главе{57}.

А сейчас мы переходим к работе с ключами на практике.

Задание 2.2.a: просмотрите ещё раз созданные при выполнении заданий 2.1.a{26} и 2.1.d{34} отношения и таблицы. Назовите их потенциальные ключи, первичные ключи, альтернативные ключи.

Задание 2.2.b: в каких отношениях базы данных «Банк»{408} стоило бы выбрать другие ключи? Внесите соответствующие правки в модель, если у вас есть достаточные аргументы для реализации таких изменений.

Задание 2.2.c: в каких отношениях базы данных «Банк»{408} стоило бы создать составные ключи? Внесите соответствующие правки в модель, если у вас есть достаточные аргументы для реализации таких изменений.

42 «Работа с MySQL, MS SQL Server и Oracle в примерах» (С.С. Куликов), пример 46 [http://svyatoslav.biz/database_book/]

2.2.2. СОЗДАНИЕ И ИСПОЛЬЗОВАНИЕ КЛЮЧЕЙ

В первую очередь стоит отметить, что далеко не все только что рассмотренные теоретические идеи находят место в повседневном применении (см. рисунок 2.2.o).

Также подчеркнём, что в данной главе речь идёт о проектировании на даталогическом и физическом уровнях (и разницы на них в плане реализации ключей практически нет). О том, что происходит с ключами на инфологическом уровне моделирования, будет рассказано в соответствующем разделе{284}.

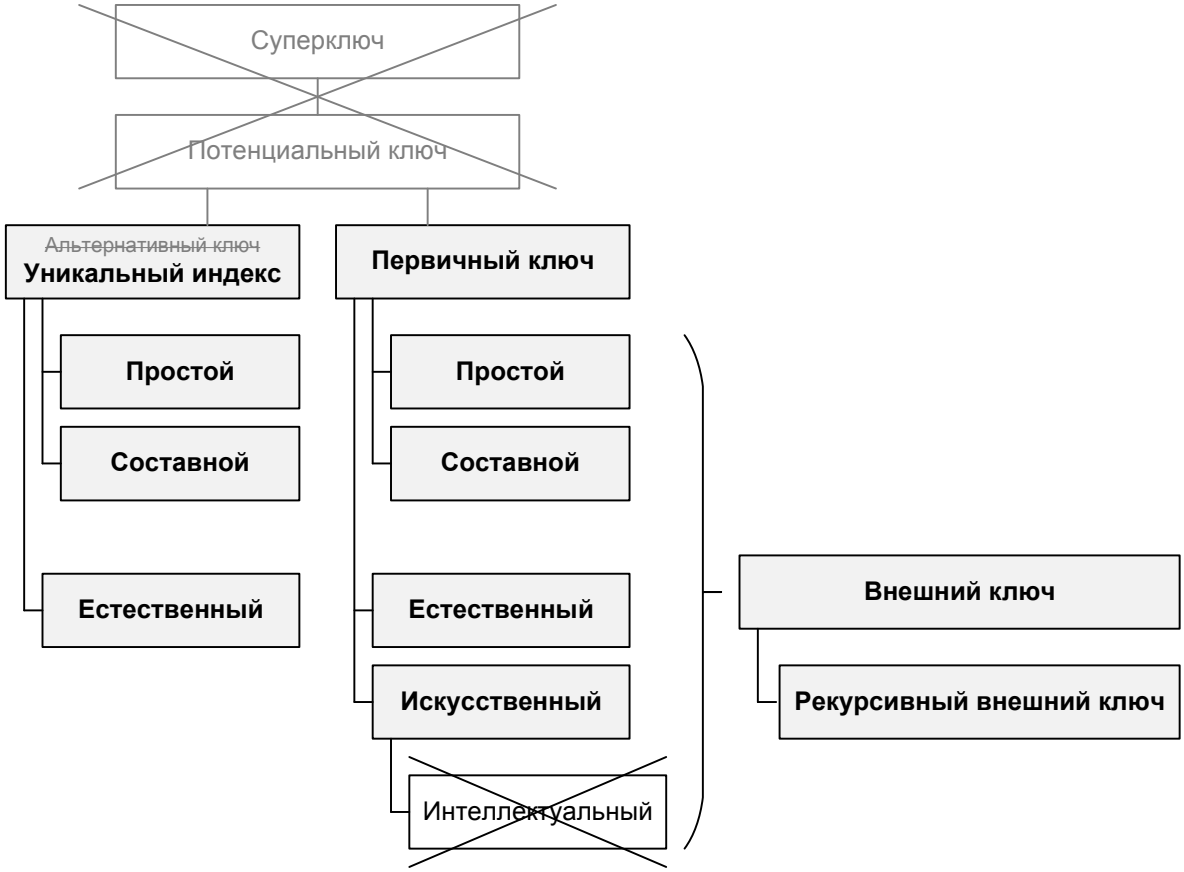


*Рисунок 2.2.o — **Виды ключей и их взаимосвязь (на практике)***

Так о суперключах почти никогда не вспоминают, а потенциальные ключи рассматривают только при принятии решения о том, как организовать в таблице первичный ключ. Альтернативные ключи превращаются в уникальные индексы{109}. А интеллектуальные ключи практически никогда не используются в силу рассмотренных выше причин{45}.

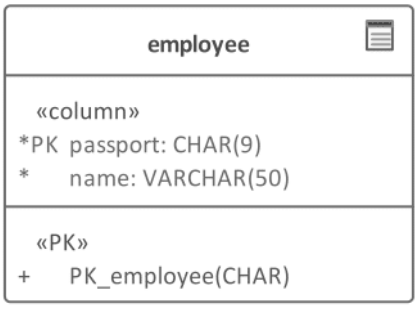
Чтобы создать простой первичный ключ таблицы, нужно:

- для естественного первичного ключа отметить соответствующим образом выбранное в качестве такого ключа поле таблицы (рисунок 2.2.p);
- для искусственного первичного ключа сначала добавить в таблицу новое поле, и затем отметить это поле соответствующим образом (рисунок 2.2.q).

Традиционно на схемах отношений первичный ключ принято размещать в самом верху, т.е. делать его первым полем таблицы (первыми полями, если ключ составной).

В подавляющем большинстве средств проектирования баз данных для «превращения» поля в первичный ключ достаточно просто отметить соответствующий чек-бокс, т.к. создание первичных ключей — крайне востребованная и очень часто выполняемая операция.

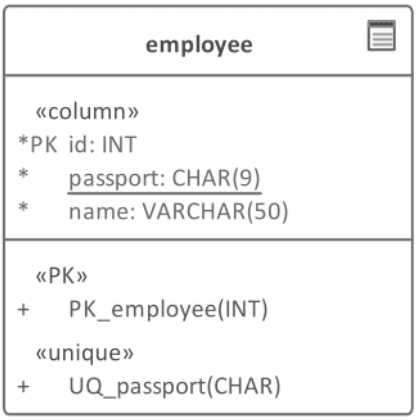
На схемах в нотации UML первичный ключ отмечается аббревиатурой «PK». Его имя иногда подчёркивается, но в общем случае подчёркивание имени атрибута означает включение контроля уникальности его значений.



MySQL Создание простого естественного первичного ключа

```
CREATE TABLE `employee`
(
 `passport` CHAR(9) NOT NULL,
 `name` VARCHAR(50) NOT NULL,
  CONSTRAINT `PK_employee`
  PRIMARY KEY (`passport`)
)
```

*Рисунок 2.2.p — **Создание простого естественного первичного ключа***



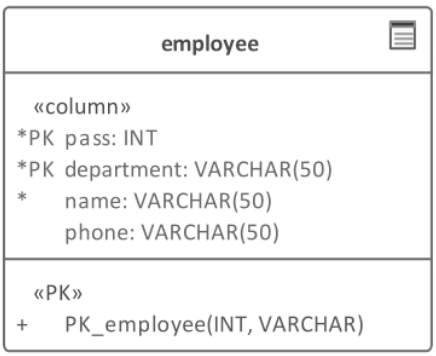
MySQL Создание искусственного первичного ключа

```
CREATE TABLE `employee`
(
 `id` INT UNSIGNED NOT NULL AUTO_INCREMENT,
 `passport` CHAR(9) NOT NULL,
 `name` VARCHAR(50) NOT NULL,
  CONSTRAINT `PK_employee`
  PRIMARY KEY (`id`)
);

ALTER TABLE `employee`
 ADD CONSTRAINT `UQ_passport`
  UNIQUE (`passport`);
```

*Рисунок 2.2.q — **Создание искусственного первичного ключа***

Обратите внимание, что в обоих отношениях первичные ключи и ограничение уникальности (`UQ_passport`) отнесены (согласно синтаксису UML) не к свойствам отношения, а к операциям. И формально это верно, т.к. то, что с точки зрения человека выглядит просто как «это поле будет первичным ключом», с точки зрения СУБД выглядит как «для этого поля впредь нужно выполнять определённый набор операций».

Чтобы создать составной естественный первичный ключ, нужно выполнить те же действия — отметить все необходимые поля как первичный ключ (рисунок 2.2.r).



MySQL Создание составного естественного первичного ключа

```
CREATE TABLE `employee`
(
 `pass` INT UNSIGNED NOT NULL AUTO_INCREMENT,
 `department` VARCHAR(50) NOT NULL,
 `name` VARCHAR(50) NOT NULL,
 `phone` VARCHAR(50) NULL,
  CONSTRAINT `PK_employee`
  PRIMARY KEY (`pass`,`department`)
)
```

*Рисунок 2.2.r — **Создание составного естественного первичного ключа***

И теперь пришло время объяснить ранее упомянутый факт{41}: последовательность полей в составном первичном ключе (равно как и в составном индексе{108}) играет очень важную роль.

Забегая вперёд скажем, что чаще всего первичный ключ является кластерным индексом{110}, т.е. СУБД физически упорядочивает данные на диске по значениям первичного ключа. Если такой ключ состоит из одного поля, никаких проблем нет.

Но если он состоит из нескольких полей, СУБД может эффективно производить поиск только по комбинации входящих в состав первичного ключа полей или по первому полю отдельно. Но по второму (третьему и т.д.) полю отдельно такой эффективный поиск не работает.

Проиллюстрируем эту мысль наглядно (рисунок 2.2.s).

Составной первичный ключ			
pass	**department**	**name**	**Phone**
1	Отд-1	Иванов И.И.	111-22-33
1	Отд-6	Петров П.П.	111-22-44
1	Отд-18	Сидоров С.С.	111-22-55
1	Отд-21	Сидоров С.С.	222-22-33
2	Отд-1	Орлов О.О.	222-22-11
2	Отд-2	Беркутов Б.Б.	222-22-00
2	Отд-3	Бобров Б.Б.	222-66-33
4	Отд-34	Синицын С.С.	999-12-11
5	Отд-1	Воробьёв В.В.	999-12-12
5	Отд-3	Воронов В.В.	999-12-15
5	Отд-7	Львов Л.Л.	999-12-17
6	Отд-4	Волков В.В.	888-10-01
7	Отд-3	Зайцев З.З.	888-10-02
7	Отд-5	Окунев О.О.	888-10-03
8	Отд-2	Карасев К.К.	888-10-05
11	Отд-3	Щукин Щ.Щ.	888-10-91
12	Отд-1	Петров П.П.	888-10-81
12	Отд-5	Сидоров С.С.	888-10-81
13	Отд-6	Сидоров С.С.	999-12-11
13	Отд-9	Орлов О.О.	999-12-12
14	Отд-17	Беркутов Б.Б.	999-12-15
18	Отд-17	Бобров Б.Б.	999-12-17
23	Отд-9	Синицын С.С.	888-10-01
23	Отд-12	Воробьёв В.В.	888-10-02
31	Отд-11	Воронов В.В.	999-12-11
32	Отд-14	Львов Л.Л.	999-12-12
34	Отд-17	Волков В.В.	999-12-15
45	Отд-1	Зайцев З.З.	999-12-17
45	Отд-2	Окунев О.О.	888-10-01
46	Отд-1	Карасев К.К.	888-10-02
56	Отд-3	Орлов О.О.	999-12-12
56	Отд-53	Беркутов Б.Б.	999-12-15
71	Отд-3	Бобров Б.Б.	999-12-17
82	Отд-1	Синицын С.С.	888-10-01

*Рисунок 2.2.s — **Иллюстрация логики работы составных первичных ключей***

Попробуйте быстро дать ответ на следующие вопросы:
- Существует ли сотрудник с номером пропуска 87?
- У какого количества сотрудников номер пропуска равен 1?
- В каком отделе работает сотрудник с номером пропуска 34?
- Как зовут сотрудника с номером пропуска 1 из от дела Отд-18?
- Какой телефон у сотрудника с номером пропуска 7 из отдела Отд-3?

Даже для человека такая задача не составляет труда: сначала вы (очень быстро) ищете по возрастанию в колонке **pass** номер пропуска и (если находите) столь же быстро по возрастанию просматриваете несколько строк колонки **department** в поисках нужного отдела. Никаких сложностей.

Теперь попробуйте столь же быстро дать ответ на следующие вопросы:
- Сколько сотрудников работает в отделе Отд-3?
- Работает ли хотя бы один сотрудник в отделе Отд-15?
- Какой самый большой номер пропуска у сотрудников из отдела Отд-2?

Эти операции выполнить столь же быстро не получается: вам приходится каждый раз полностью (с самого верха до самого низа) просматривать всю колонку **department** и при нахождении каждого искомого совпадения временно переключаться на другие действия.

Пускай в реальности СУБД выполняет несколько иные операции, концепция остаётся той же: по совокупности полей ключа или отдельно по первому полю ключа поиск выполняется очень быстро, а отдельно по второму (третьему и т.д.) полю ключа поиск выполняется медленно.

Отсюда следует очень простой вывод: если при работе с вашей базой данных часто будут выполняться запросы, в которых поиск происходит по отдельному полю составного ключа, это поле должно быть первым в составе такого ключа.

С первичными ключами на этом всё. Переходим к уникальным индексам{109}, в которые «превратились» альтернативные ключи.

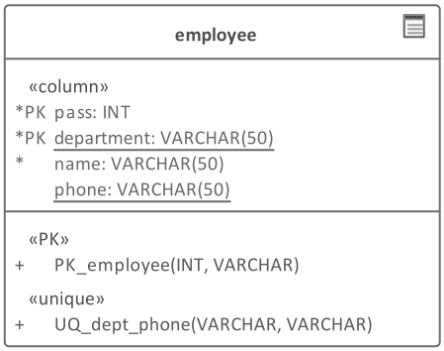
Технически создание уникальных индексов в большинстве средств проектирования баз данных выполняется отметкой свойства **unique** у соответствующего поля таблицы. В некоторых средствах проектирования придётся создать у таблицы отдельную операцию (с типом **unique**) и указать поле (или поля), с которым она должна работать.

Как создать простой естественный уникальный индекс, уже было (пусть и неявно) показано выше — на рисунке 2.2.q такой индекс создан на поле **passport**, чтобы гарантировать уникальность его значений.

Если необходимо создать составной естественный уникальный индекс, операции типа **unique** нужно указать весь набор полей, с которыми она должна будет работать (здесь уже недостаточно просто отметить у каждого поля свойство unique, т.к. средство проектирования может посчитать, что вы создаёте не один составной индекс{108}, а много отдельных простых индексов{108}).

Допустим, по неким бизнес-правилам в некоей компании запрещено, чтобы в одном и том же отделе у двух и более сотрудников были одинаковые номера телефонов (т.е. комбинация значений поле **department** и **phone** должна быть уникальной).

На рисунке 2.2.t показана соответствующая ситуация (помните про последовательность полей{52} — для индексов характерны все те же рассуждения, что были приведены для ключей).



MySQL **Создание составного естественного уникального индекса**

```
CREATE TABLE `employee`
(
 `pass` INT UNSIGNED NOT NULL AUTO_INCREMENT,
 `department` VARCHAR(50) NOT NULL,
 `name` VARCHAR(50) NOT NULL,
 `phone` VARCHAR(50)     NULL,
  CONSTRAINT `PK_employee`
  PRIMARY KEY (`pass`,`department`)
);

ALTER TABLE `employee `
 ADD CONSTRAINT `UQ_dept_phone`
  UNIQUE (`department`,`phone`);
```

*Рисунок 2.2.t — **Создание составного естественного уникального индекса***

Пусть вас не смущает, что поле **`department`** теперь входит как в состав первичного ключа, так и в состав уникального индекса. В общем случае (пока мы не начали говорить о нормализации{161}) это вполне типичная и допустимая ситуация.

Интересен и ещё один факт: с появлением индекса {**`department`**, **`phone`**} (где поле **`department`** идёт первым) СУБД получила возможность быстрого поиска и отдельно по полю **`department`**, т.е. ранее озвученные «вопросы, на которые не получается дать быстрый ответ{53}» теперь для СУБД перестали быть проблемой.

С первичными ключами и уникальными индексами почти всё. Если вас всё же интересует вопрос о том, можно ли современные СУБД заставить использовать интеллектуальный ключ, то — да, можно (хоть и не нужно, т.к. скорее всего проблем будет больше, чем пользы): классическим решением будет использование триггеров{350} (которые будут формировать значение ключа) или индексов на вычисляемых столбцах{129}, если выбранная вами СУБД их поддерживает.

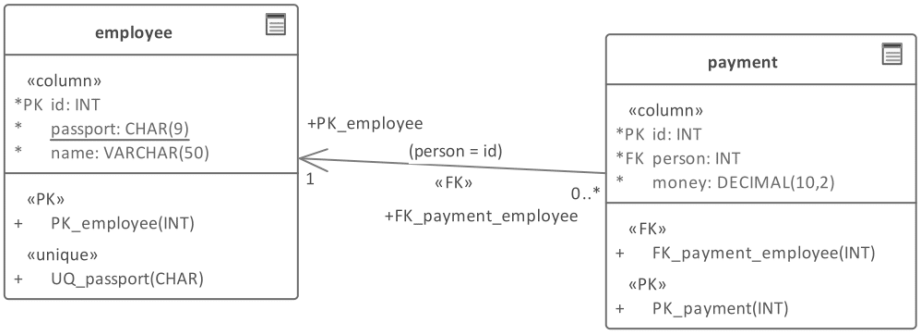
Осталось рассмотреть создание внешних ключей. Этот вопрос тесно затрагивает понятие «связи», и потому рекомендуется ознакомиться в т.ч. с соответствующей главой{57}.

СУБД начинает «понимать», что некое поле (или группа полей) таблицы является внешним ключом только в тот момент, когда между таблицами явно устанавливается связь. Потому для создания внешнего ключа надо провести связь между соответствующими дочерней и родительской таблицами (рисунок 2.2.u).

Важно! Направление связи имеет значение. Именно оно определяет, какая таблица является дочерней, а какая — родительской. Потому обязательно прочитайте в документации к вашему средству проектирования, в каком направлении (от родительской таблицы к дочерней таблице или от дочерней таблицы к родительской таблице) в нём нужно проводить связи.

Отметим, что большинство средств проектирования придерживается правила «от дочерней к родительской».

Ещё одна сложность состоит в том, что некоторые средства проектирования автоматически создают в дочерней таблице поле (или группу полей), которое будет внешним ключом, некоторые предполагают, что это поле уже создано (и предлагают его выбрать), а некоторые и вовсе умеют работать обоими способами. Опять же: читайте документацию!

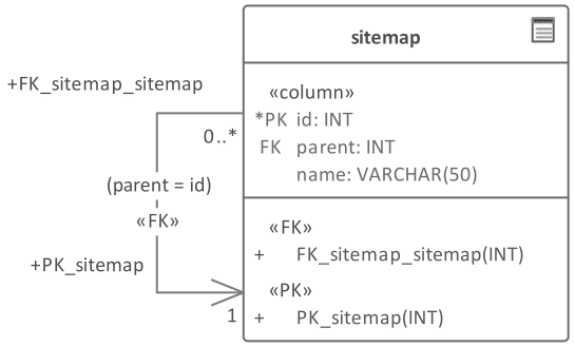


MySQL **Создание внешнего ключа (и соответствующей связи)**

```
CREATE TABLE `employee`
(
      `id` INT UNSIGNED NOT NULL AUTO_INCREMENT,
      `passport` CHAR(9) NOT NULL,
      `name` VARCHAR(50) NOT NULL,
      CONSTRAINT `PK_employee` PRIMARY KEY (`id`)
);

CREATE TABLE `payment`
(
      `id` INT UNSIGNED NOT NULL AUTO_INCREMENT,
      `person` INT UNSIGNED NOT NULL,
      `money` DECIMAL(10,2) NOT NULL,
      CONSTRAINT `PK_payment` PRIMARY KEY (`id`)
);

ALTER TABLE `employee`
 ADD CONSTRAINT `UQ_passport` UNIQUE (`passport`);

ALTER TABLE `payment`
 ADD CONSTRAINT `FK_payment_employee`
  FOREIGN KEY (`person`) REFERENCES `employee` (`id`)
   ON DELETE CASCADE ON UPDATE RESTRICT;
```

*Рисунок 2.2.u — **Создание внешнего ключа (и соответствующей связи)***

При создании рекурсивного внешнего ключа связь возвращается в ту же таблицу, откуда она начинается (рисунок 2.2.v).



MySQL **Создание рекурсивного внешнего ключа (и соответствующей связи)**

```
CREATE TABLE `sitemap`
(
  `id` INT UNSIGNED
       NOT NULL AUTO_INCREMENT,
  `parent` INT UNSIGNED NULL,
  `name` VARCHAR(50) NULL,
   CONSTRAINT `PK_sitemap`
   PRIMARY KEY (`id`)
);

ALTER TABLE `sitemap`
 ADD CONSTRAINT
  `FK_sitemap_sitemap`
   FOREIGN KEY (`parent`)
   REFERENCES `sitemap` (`id`)
   ON DELETE CASCADE
   ON UPDATE RESTRICT;
```

*Рисунок 2.2.v — **Создание рекурсивного внешнего ключа (и соответствующей связи)***

И остаётся подчеркнуть ещё два важных момента, часто неочевидных для тех, кто только начинает работать с базами данных.

В UML (и многих других нотациях) стрелки связей идут просто от одного отношения к другому, а не от конкретного поля к конкретному полю. Т.е. бессмысленно пытаться понять, по каким полям связаны отношения, высматривая «место, откуда выходит и куда приходит стрелка связи». Соответствующая информация написана на самой связи, а геометрия линий определяется исключительно удобством расположения на схеме.

Критически важно! Типы данных первичного ключа и соответствующих внешних ключей должны быть полностью идентичными (т.е. тип, размер, «знаковость», кодировка, формат и т.д. — любые подобные свойства, которые есть у соответствующих полей, должны совпадать (кроме автоинкрементируемости, которой не должно быть у внешнего ключа)). Это важно потому, что в такой ситуации СУБД не тратит дополнительные ресурсы на преобразование данных при сравнении значений первичного и внешнего ключа. Также существует рекомендация создавать на внешних ключах индексы{106}, что может ускорить выполнение `JOIN`-запросов.

Почти все современные СУБД позволяют очень грубо нарушать это правило (допустим, сделать первичный ключ `BIGINT`-числом, а внешний ключ — `VARCHAR`-строкой), но такое издевательство над базой данных как минимум выливается в катастрофическое падение производительности, а как максимум и вовсе к ошибкам при выполнении некоторых запросов.

В завершении данной главы ещё раз отметим, что на даталогическом и физическом уровнях моделирования работа с ключами выглядит практически идентично.

Из возможных различий можно упомянуть лишь более широкий спектр свойств ключа, доступный на физическом уровне в некоторых средствах проектирования (хотя чаще всего таких различий нет вовсе).

На этом рассмотрение ключей закончено, но в следующих разделах (см. практическую реализацию связей{78} и триггеров{350}) очень многие прозвучавшие здесь мысли будут продолжены и представлены более подробно.

Задание 2.2.d: изучите в документации к выбранной вами СУБД различные способы создания ключей (как правило, существует несколько альтернативных синтаксических форм, позволяющих получить один и тот же результат).

Задание 2.2.e: если при выполнении задания 2.2.c{49} вы предложили использование составных ключей, оптимально ли в них расположены элементы? Внесите соответствующие правки в модель, если у вас есть достаточные аргументы для реализации таких изменений.

Задание 2.2.f: можно ли в коде по созданию базы данных «Банк»{412} использовать альтернативный синтаксис по формированию ключей? Если вы считаете, что «да», напишите такой код и проверьте его работоспособность.

2.3. СВЯЗИ

2.3.1. ОБЩИЕ СВЕДЕНИЯ О СВЯЗЯХ

В данной главе нам понадобится два основных определения, на основе которых строятся все последующие рассуждения.

Связь (relationship[43]) — ассоциация, объединяющая несколько сущностей.

Упрощённо: указание того факта, что отношения находятся в некоей взаимосвязи друг с другом.

Мощность связи (relationship cardinality[44]) — свойство связи, определяющее допустимые мощности взаимосвязанных подмножеств кортежей отношений, объединённых данной связью.

Упрощённо: указание количества взаимосвязанных строк в таблицах, объединённых связью (например: «один ко многим», «многие ко многим», «один к трём» и т.д.)

Как и было сказано ранее, связи организуются с использованием внешних ключей{47}. Причём для случая «многие ко многим» помимо внешних ключей необходимо создать т.н. «таблицу связи» (см. далее{60}).

Теперь рассмотрим три классических вида связей.

Связь один ко многим (one to many correspondence[45]) — ассоциация, объединяющая два отношения таким образом, что одному кортежу родительского отношения (или даже нулю таких кортежей) может соответствовать произвольное количество кортежей дочернего отношения.

Связь многие к одному (many to one correspondence[46]) — ассоциация, объединяющая два отношения таким образом, произвольное количество кортежей дочернего отношения может соответствовать одному кортежу родительского отношения (или даже нулю таких кортежей).

Упрощённо: одной записи в таблице A может соответствовать много записей в таблице B; при этом может быть ситуация, когда неким записям в таблице B нет соответствия в таблице A.

Эти два определения не зря приведены вместе. Обратите внимание: англоязычные определения этих двух ситуаций полностью идентичны (и это — не опечатка). Единственная «разница» заключается в том, как (в каком направлении) анализировать связь.

---

43 **Relationship** — an association among entities. («The New Relational Database Dictionary», C.J. Date)

44 **Cardinality** — the number of elements in a set. («The New Relational Database Dictionary», C.J. Date)

45 **One to many correspondence** — a rule pairing two sets s1 and s2 (not necessarily distinct) such that each element of s1 corresponds to at least one element of s2 and each element of s2 corresponds to exactly one element of s1; equivalently, that pairing itself. («The New Relational Database Dictionary», C.J. Date)

46 **Many to one correspondence** — a rule pairing two sets s1 and s2 (not necessarily distinct) such that each element of s1 corresponds to exactly one element of s2 and each element of s2 corresponds to at least one element of s1; equivalently, that pairing itself. («The New Relational Database Dictionary», C.J. Date)

Рассмотрим на примере (рисунки 2.3.a и 2.3.b; также см. практическую реализацию в соответствующем разделе{78}). Если начинать анализ связи со стороны отношения **`employee`**, то мы получим связь «один ко многим», т.е. «одному сотруднику принадлежит много платежей». Если начинать анализ связи со стороны отношения **`payment`**, то мы получим связь «многие к одному», т.е. «много платежей принадлежит одному сотруднику». Но на самом деле это — одна и та же связь.

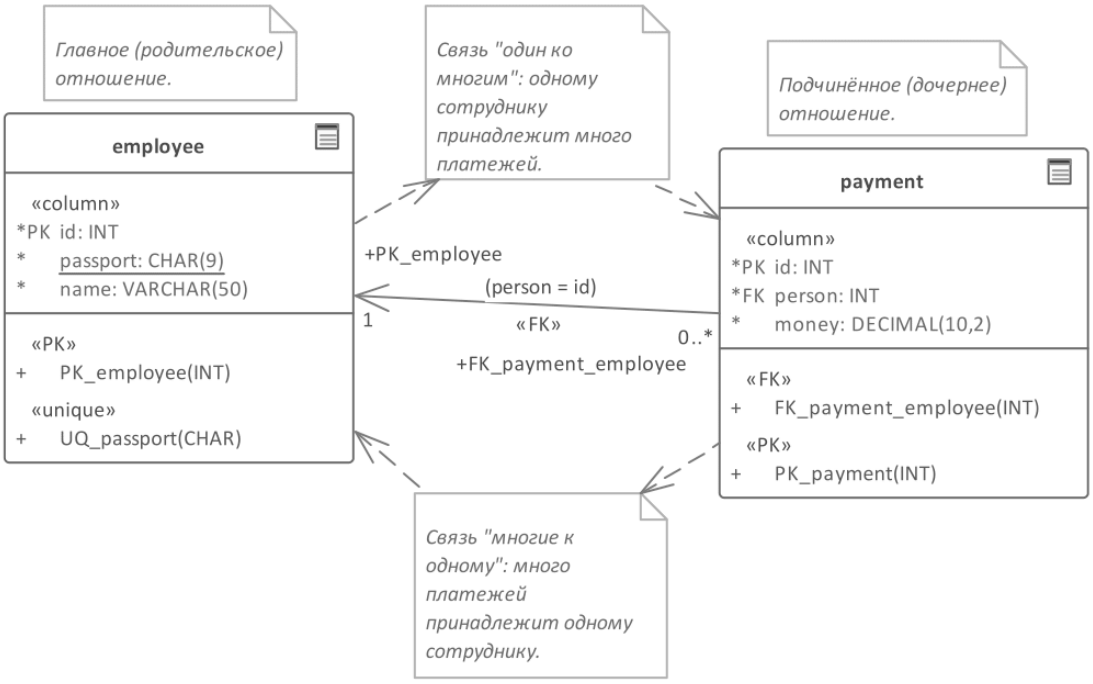


*Рисунок 2.3.a — **Схема, демонстрирующая связь «один ко многим» (она же — «многие к одному»)***

*Рисунок 2.3.b — **Таблицы, демонстрирующие связь «один ко многим» (она же — «многие к одному»)***

Важный вывод: в реальности (на уровне СУБД) не существует никакой разницы между связями «один ко многим» и «многие к одному». Для простоты восприятия практически везде эта связь называется «один ко многим» (начинать анализ со стороны родительской таблицы проще и привычнее).

Для простоты запоминания: главная (родительская) таблица — та, со стороны которой «один», а подчинённая (дочерняя) таблица — та, со стороны которой «много».

Для организации данной связи необходимо использовать внешний ключ{47}, который находится в дочерней таблице. В дочерней же таблице отражаются дополнительные свойства связи.

Продолжим пример, показанный на рисунке 2.3.b и предположим, что платежи бывают наличными (cash) и безналичными (wire). Т.е. недостаточно просто показать, что «Иванову начислена такая-то сумма», нужно показать, «как именно» эта сумма начислена.

Это «как именно» в данном случае и является свойством связи, которое в случае связи «один ко многим» отражается в дочерней таблице (см. рисунок 2.3.c). Объективно это свойство нельзя отразить в родительской таблице, т.к. тогда получилось бы, что некоему сотруднику можно платить **или** только наличным, **или** только безналичным способом.

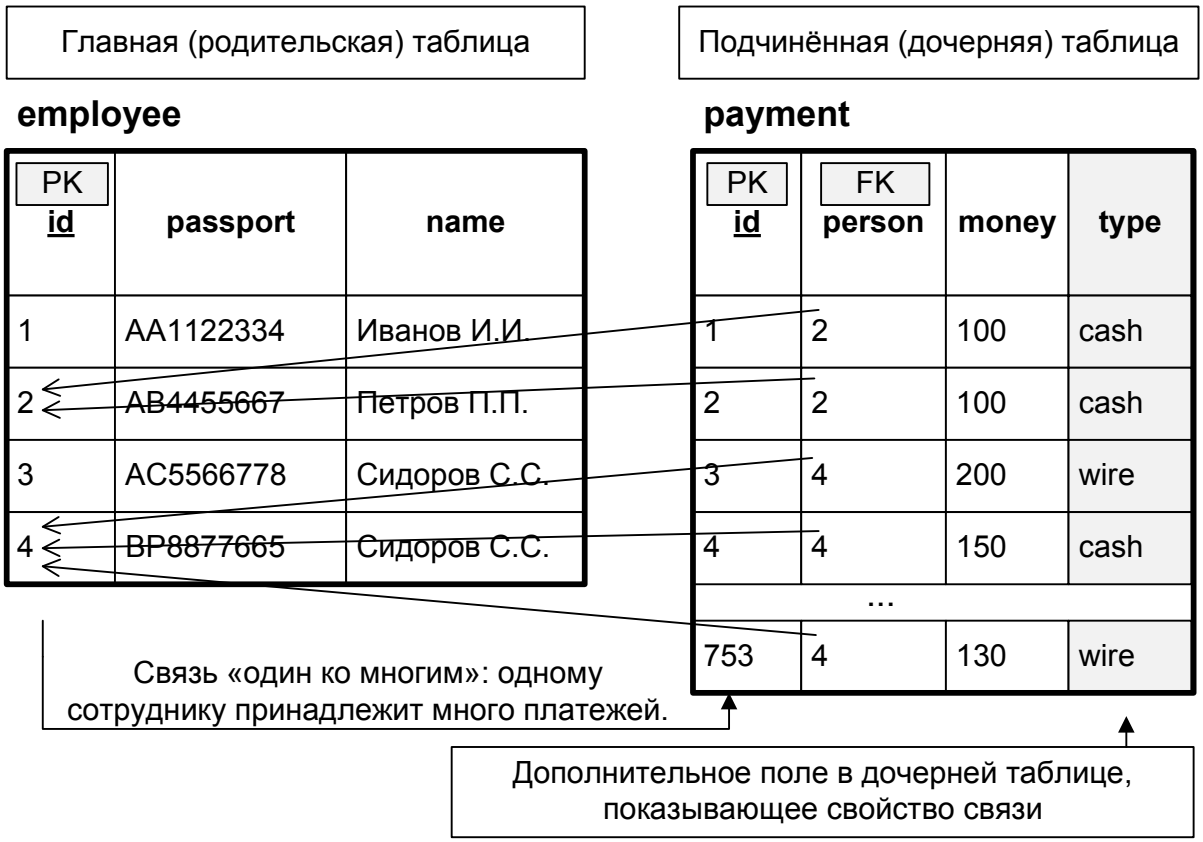


*Рисунок 2.3.c — **Способ отражения свойств связи «один ко многим»***

Если у связи появятся и другие свойства (например, был ли платёж авансовым, начислялся ли он на основной счёт или дополнительный, в какой валюте он начислялся и т.д.) — все они будут выражены дополнительными полями в дочерней таблице.

Для простоты запоминания можно считать, что в случае связи «один ко многим» свойства связи эквивалентны свойствам записей в дочерней таблице.

Связь многие ко многим (many to many correspondence[47]) — ассоциация, объединяющая два отношения таким образом, что одному кортежу любого из объединённых отношений может соответствовать произвольное количество кортежей второго отношения.

Упрощённо: одной записи в таблице A может соответствовать много записей в таблице B, и в то же время одной записи в таблице B может соответствовать много записей в таблице A.

Рассмотрим следующий классический пример: студенты и предметы. Каждый студент изучает много предметов, и каждый предмет изучается многими студентами.

Отразить такую ситуацию в базе данных с помощью связи «один ко многим» невозможно, т.к. здесь нет явной родительской и подчинённой сущностей (и попытка выполнить взаимную миграцию первичных ключей приведёт к очень серьёзным техническим нарушениям и полной потере адекватности предметной области{12}).

В таком случае создаётся т.н. «таблица связи» (associacion relation) (появляется она в большинстве средств проектирования на физическом уровне), которая является дочерней по отношению к обеим связываемым таблицам. И именно в эту таблицу связей мигрируют первичные ключи связываемых таблиц.

На даталогическом уровне (см. рисунок 2.3.d) у такой связи даже не указываются такие технические параметры, как ключи, операции и т.д. — их здесь пока просто нет[48] (они могут появиться только вместе с таблицей связи).

На физическом же уровне (см. рисунки 2.3.e и 2.3.f; также см. практическую реализацию в соответствующем разделе{88}) мы получаем, фактически, две отдельных связи «один ко многим», через которые и формируется нужная нам связь «многие ко многим».

При этом в таблице связей получается два внешних ключа, которые чаще всего объединяются в составной естественный первичный ключ (необходимость этой операции будет рассмотрена далее{89}).

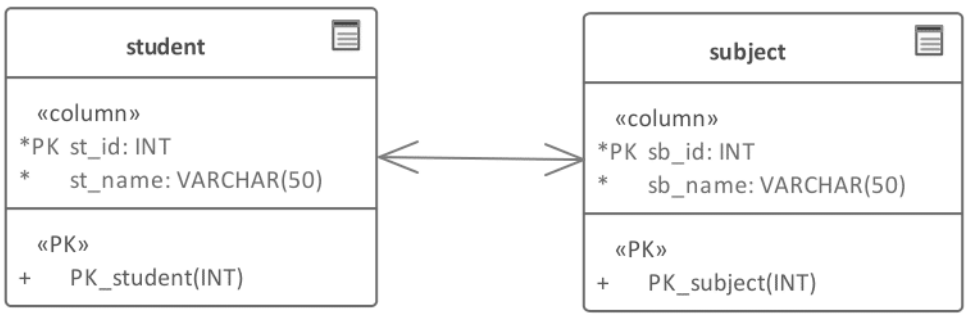


*Рисунок 2.3.d — **Схема, демонстрирующая связь «многие ко многим» на даталогическом уровне***

---

47 **Many to many correspondence** — a rule pairing two sets s1 and s2 (not necessarily distinct) such that each element of s1 corresponds to at least one element of s2 and each element of s2 corresponds to at least one element of s1; equivalently, that pairing itself. («The New Relational Database Dictionary», C.J. Date)

48 Да, часто эта информация появляется в процессе моделирования даталогического уровня, хотя формально она и относится к физическому: так просто удобнее, и это ещё раз иллюстрирует тот факт, что границы между уровнями моделирования носят весьма условный характер.

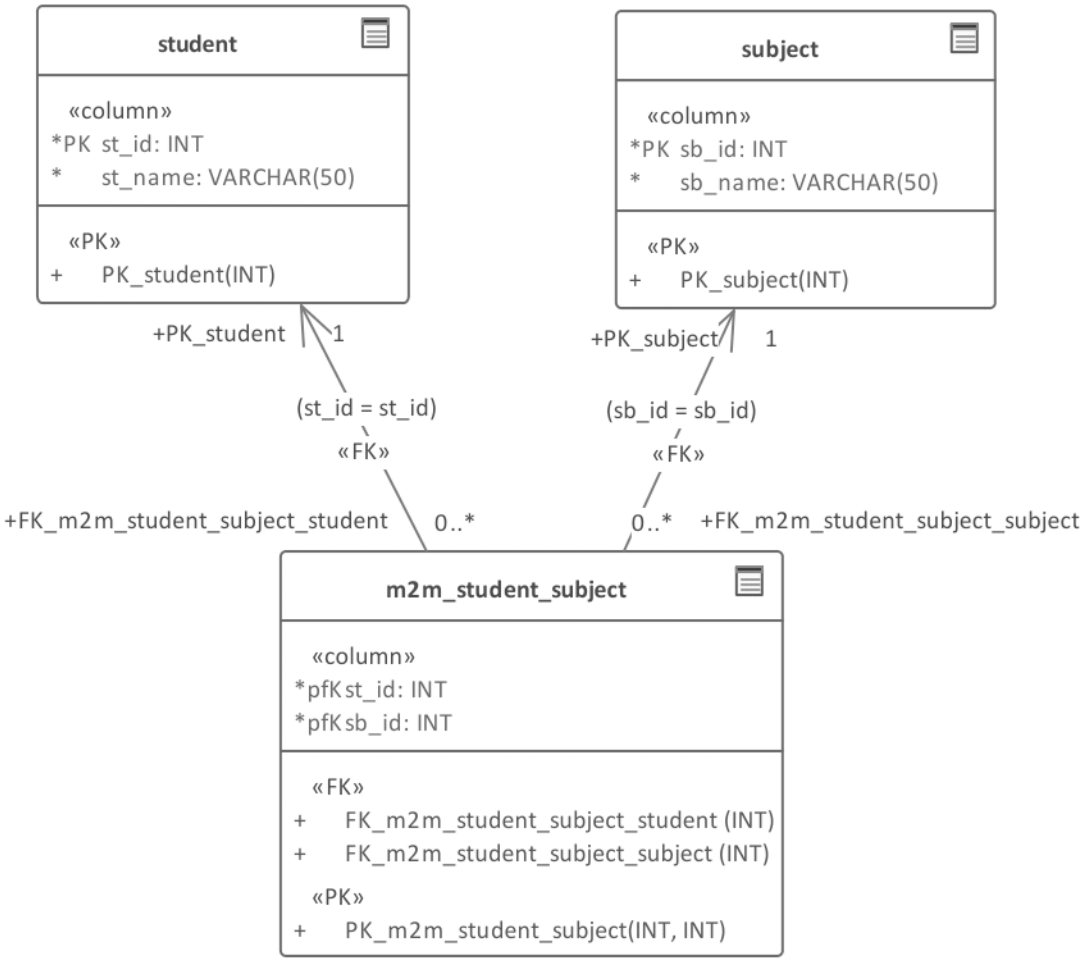


*Рисунок 2.3.e — **Схема, демонстрирующая связь «многие ко многим» на физическом уровне***

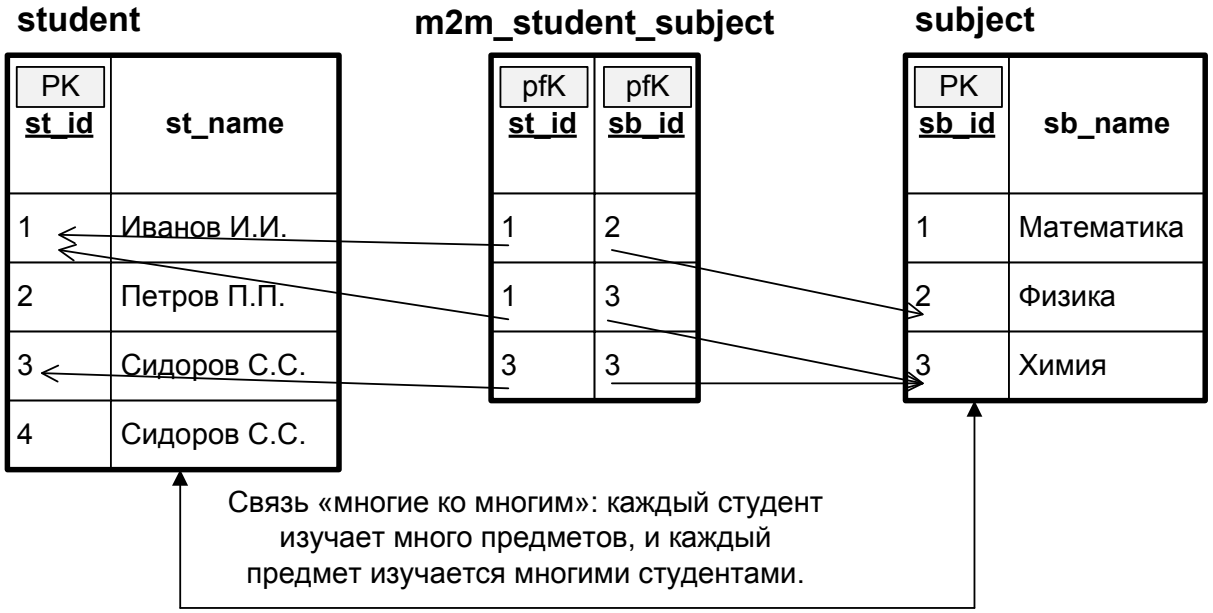


*Рисунок 2.3.f — **Таблицы, демонстрирующие связь «многие ко многим» на физическом уровне***

В случае связи «многие ко многим» т.н. «свойство связи» отражается в таблице связи. Например, по итогам изучения предмета студент должен получить оценку. Но оценка объективно не является ни свойством студента (у него много оценок, а не одна), ни свойством предмета (предмету вообще нелогично ставить оценки) — она является именно свойством взаимосвязи конкретного студента и конкретного предмета. Соответствующий пример представлен на рисунке 2.3.g.

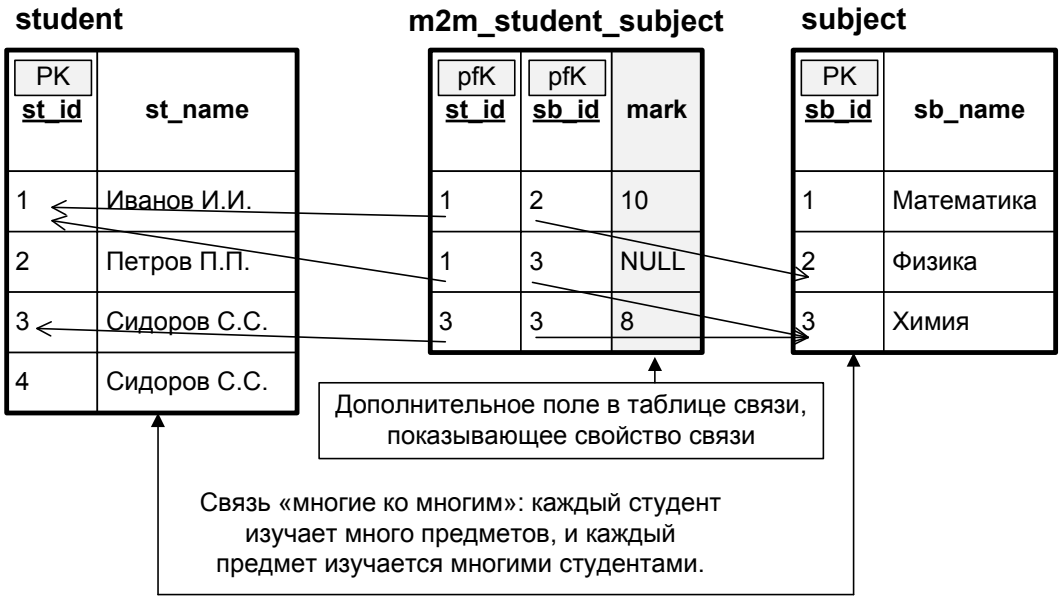


*Рисунок 2.3.g — **Способ отражения свойств связи «многие ко многим»***

И остаётся рассмотреть последний классический вариант связей.

Связь один к одному (one to one correspondence[49]) — ассоциация, объединяющая два отношения таким образом, что одному кортежу родительского отношения может соответствовать не более одного кортежа дочернего отношения.

Упрощённо: одной записи в таблице A может соответствовать не более одной записи в таблице B.

Из определения следует, что эта связь является частным случаем связи «один ко многим», где на дочернее отношение наложено дополнительное ограничение — внешний ключ в таком отношении должен обладать свойством уникальности своих значений (иногда это достигается за счёт того, что внешний ключ одновременно является первичным, но существуют и другие технические способы, например, использование уникального индекса{109}).

Мы рассматриваем эту связь в последнюю очередь потому, что у неё очень узкая область применения.

---

[49] **One to one correspondence** — a rule pairing two sets s1 and s2 (not necessarily distinct) such that each element of s1 corresponds to exactly one element of s2 and each element of s2 corresponds to exactly one element of s1; equivalently, that pairing itself. («The New Relational Database Dictionary», C.J. Date)

Если при проектировании базы данных у вас появляется связь «один к одному», то или у вас есть очень весомые аргументы в пользу её существования, или… у вас ошибка в проектировании. «Сами по себе» такие связи появляются крайне редко, т.к. ничто не мешает просто поместить все соответствующие атрибуты в одно отношение (вместо двух, объединённых связью «один к одному»).

Итак, связи «один к одному» используются, если:

- в предметной области действительно есть сущности разных типов, объединённые именно такой связью (например, «водительские права» и «талон фиксации нарушений ПДД»; хотя даже в этой ситуации очень велика вероятность помещения всей информации в одно отношение);
- при описании некоторой сущности с огромным количеством свойств мы достигли ограничения СУБД на количество полей в одной таблице или на максимальный размер записи — тогда можно «продолжить» в следующей таблице, объединив её с предыдущей этой связью (но сразу возникает сомнение в том, что модель построена правильно, т.к. в реальности такая ситуация крайне маловероятна);
- в целях оптимизации производительности мы хотим разделить на две отдельных таблицы те данные, к которым доступ нужен очень часто, и те, к которым доступ нужен очень редко, что позволит СУБД обрабатывать при частых операциях меньшие объёмы данных (и, опять же, существуют другие способы оптимизации производительности для таких ситуаций);
- в описываемой предметной области есть много разнотипных сущностей, у каждой из которых, тем не менее, есть одинаковый набор совпадающих свойств (пожалуй, это — единственный случай, в котором без связи «один к одному» не обойтись; будет рассмотрено далее{91}).

На схеме (см. рисунок 2.3.h) связь «один к одному» выглядит почти как «один ко многим» (отличие заключается в числах, указывающих мощность связи{57}, и в том, что в дочерней таблице первичный ключ одновременно является и внешним).

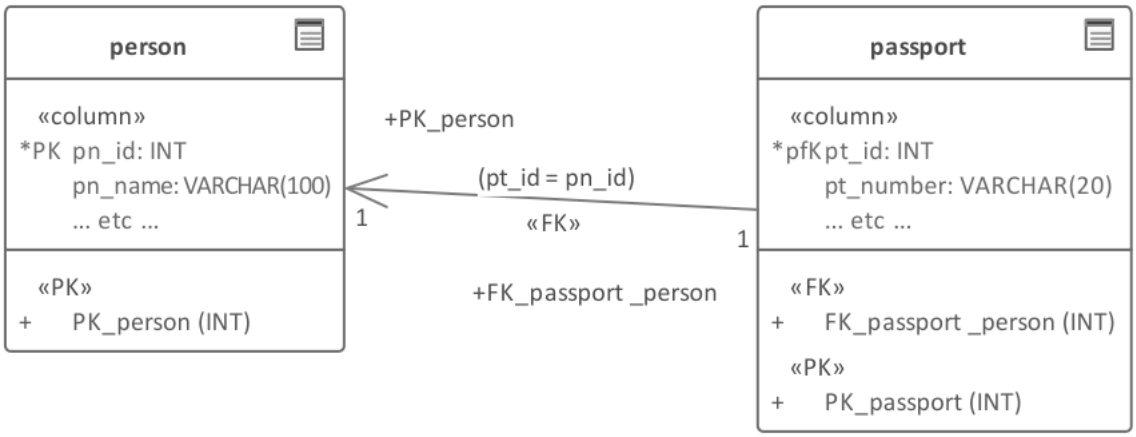


*Рисунок 2.3.h — **Схема, демонстрирующая связь «один к одному»***

Более подробные примеры по работе со связями «один к одному» будут показаны в соответствующем разделе{91}.

Мощность связи может принимать и «нестандартное» значение, т.е. кроме связей «один ко многим», «один к одному» и «многие ко многим» могут существовать связи вида «X к Y», где X и Y — произвольные целые числа или диапазоны значений, например, «1 к 3», «1 к 0…5», «0…2 к 1» и т.д.

Все соответствующие варианты определяются исключительно требованиями предметной области, которую описывает база данных. Например:

- «в системе обязательно должен быть хотя бы один администратор, но их может быть не более трёх» → «1 к 1…3»;
- «у сотрудника может не быть ни одного специального пропуска, но если такие пропуска есть, их не может быть больше пяти» → «1 к 0…5»;
- «велосипед может быть никем не арендован, и каждый человек может арендовать не более десяти велосипедов» → «0…1 к 0…10»;
- «за каждый сервер может отвечать от одного до трёх системных администраторов, но каждый администратор может отвечать не более, чем за двадцать серверов» → «1…3 к 0…20».

Технически[50] контроль за соблюдением ограничения мощности связей в таком случае реализуется с помощью триггеров{350}. Однако не менее часто из соображений удобства использования базы данных такие ограничения реализуют не в ней, а в приложениях, работающих с ней. Особенно это актуально в случае, если для разных приложений соответствующие ограничения различаются, или же эти ограничения могут меняться во времени (допустим, главный системный администратор может произвольно менять максимальное количество серверов, за которое отвечают его подчинённые).

Только что мы рассматривали случаи, когда мощность связи выражается диапазоном значений (например, «1 к 0…5»). Подобная ситуация характерна и для «классических» мощностей связей, которые могут быть представлены так:

- связь «один ко многим»: «1 к 1…M», «1 к 0…M», «0…1 к 1…M», «0…1 к 1…M» и т.д.
- связь «многие ко многим»: «1…M к 1…N», «0…M к 1…N», «1…M к 0…N», «0…M к 0…N» и т.д.
- связь «один к одному»: «1 к 1», «1 к 0…1», «0…1 к 1», «0…1 к 0…1».

Из этого явления у связей, для которых актуально понятие «родительской» и «дочерней» таблиц (т.е. связей «один ко многим» и «один к одному»), появляется ещё одно свойство — они могут быть идентифицирующими и неидентифицирующими.

Идентифицирующая связь (identifying relationship[51]) — ассоциация, объединяющая два отношения таким образом, что любому кортежу дочернего отношения всегда сопоставлен кортеж родительского отношения, а сам кортеж дочернего отношения может быть полностью определён только в совокупности со значением первичного ключа соответствующего кортежа родительского отношения.

Упрощённо: запись в дочерней таблице не может существовать без соответствующей записи в родительской таблице (для гарантированного обеспечения этого свойства внешний ключ делают частью составного первичного ключа дочерней таблицы); в случае последовательной связи в любом дочернем отношении хранятся данные обо всех родительских отношениях.

---

50 «Работа с MySQL, MS SQL Server и Oracle в примерах» (С.С. Куликов), пример 33 [http://svyatoslav.biz/database_book/]

51 **Identifying relationship** — the relationship type that relates a weak entity type to its owner. («Fundamentals of Database Systems», Ramez Elmasri, Shamkant Navathe)

Неидентифицирующая связь (non-identifying relationship[52]) — ассоциация, объединяющая два отношения таким образом, что кортежу дочернего отношения может не соответствовать ни одного кортежа родительского отношения, а сам кортеж дочернего отношения может быть полностью определён без использования значения первичного ключа соответствующего кортежа родительского отношения.

Упрощённо: запись в дочерней таблице может существовать без соответствующей записи в родительской таблице; в случае последовательной связи в любом дочернем отношении хранятся данные только об одном (ближайшем) родительском отношении.

См. подробное описание логики происхождения и применения таких связей в книге «Fundamentals of Database Systems» (Ramez Elmasri, Shamkant Navathe), 6-е издание, глава 7.5 «Weak entity types».

В качестве примера использования идентифицирующей связи рассмотрим ситуацию с проведением учебных курсов в некоем тренинговом центре. Каждый курс может проводиться много раз, потому на схеме базы данных эту часть предметной области можно выразить связью типа «один ко многим», а поскольку каждый факт проведения курса обязан быть строго привязан к соответствующему курсу и может быть определён только через этот курс, связь будет идентифицирующей.

Соответствующая схема и пример с данными показаны на рисунках 2.3.i и 2.3.j.

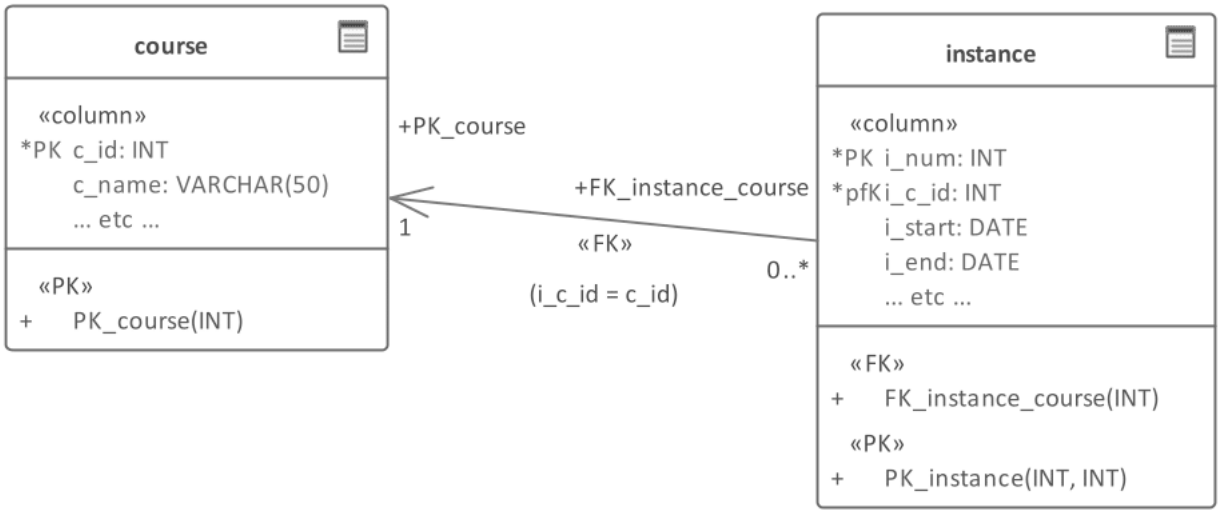


*Рисунок 2.3.i — **Схема, демонстрирующая идентифицирующую связь***

[52] **Non-identifying relationship** — a regular association (relationship) between two independent classes. («Fundamentals of Database Systems», Ramez Elmasri, Shamkant Navathe)

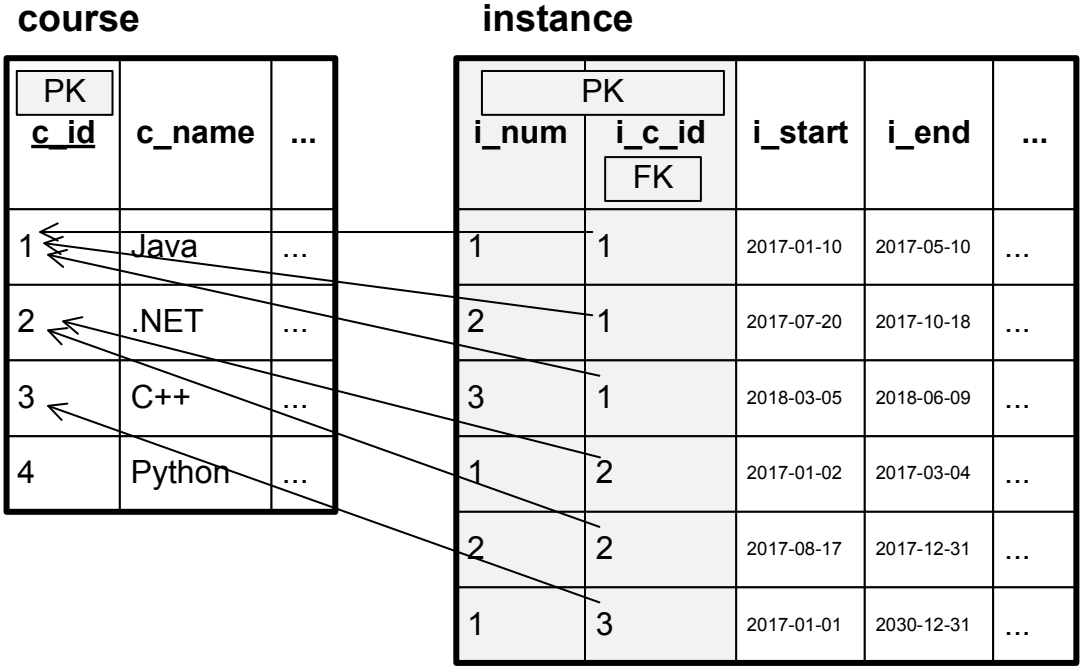


*Рисунок 2.3.j — **Пример данных, демонстрирующий идентифицирующую связь***

Итак, для создания идентифицирующей связи необходимо внешний ключ дочернего отношения сделать частью его же первичного ключа, т.е. первичный ключ дочернего отношения в случае идентифицирующей связи (почти[53]) всегда будет составным (и будет включать в себя весь первичный ключ родительского отношения, даже если тот в свою очередь также является составным).

Существует важная техническая особенность, которую стоит учитывать при формировании идентифицирующих связей: т.к. в их классической реализации внешний ключ входит в состав первичного, мы получаем уже знакомую ситуацию составного ключа{40}, в котором важен порядок следования полей{52}. Внешний ключ стоит поставить на первое место, что позволит избежать необходимости создания на нём отдельного индекса.

Необходимость добавлять в первичный ключ дочернего отношения первичный ключ родительского отношения в случае нескольких последовательных идентифицирующих связей «один ко многим» приводит к тому, что у отношений, стоящих «в конце этой цепи» первичные ключи могут состоять из большого количества полей (см. рисунок 2.3.k). С точки зрения производительности, оптимальности хранения данных и удобства работы с такой базой данных эта ситуация вызывает очень много вопросов. Именно поэтому на практике классические идентифицирующие связи используются крайне редко.

---

[53] Кроме случаев связи «один к одному», где внешний ключ, состоящий из одного поля, может сам же являться и первичным ключом.

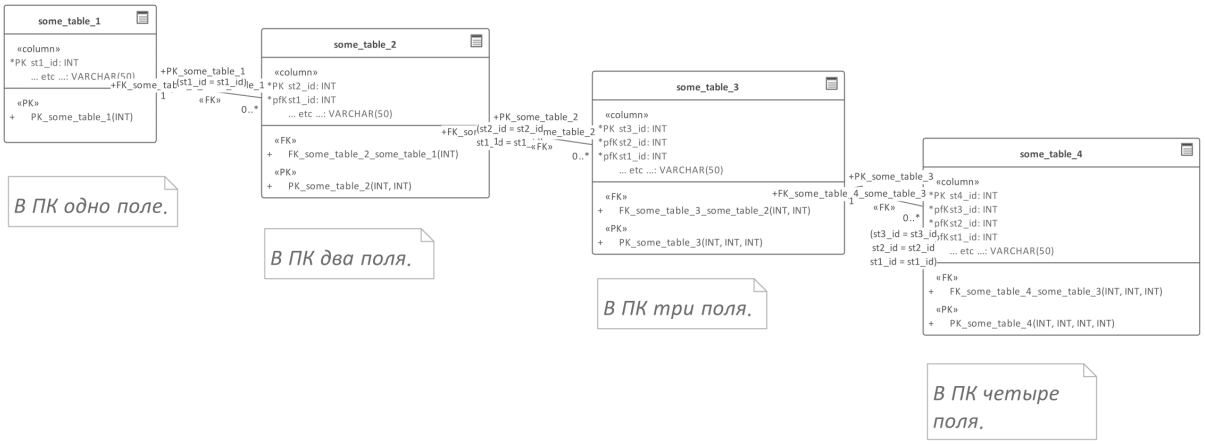


*Рисунок 2.3.k — **Проблема увеличения размера первичных ключей при использовании идентифицирующих связей***

Однако у такого «накопления ключей» есть и положительная сторона: в любой момент времени мы можем «двигаться по цепи» в любом направлении на любое количество шагов без промежуточных операций. Рассмотрим это на примере (рисунок 2.3.l).

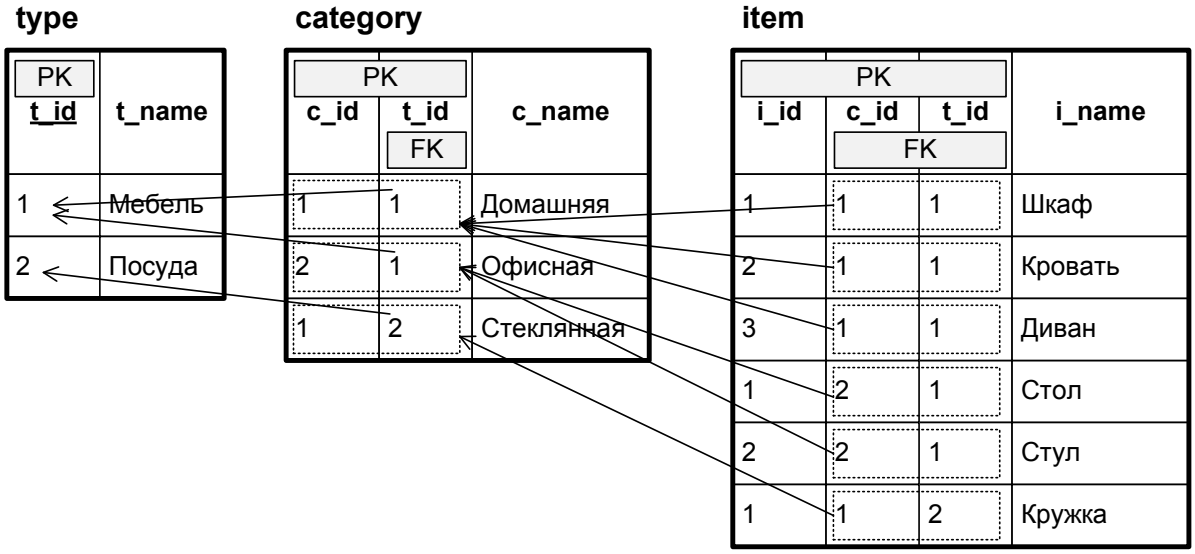


*Рисунок 2.3.l — **Пример данных для случая использования нескольких последовательных идентифицирующих связей***

Чтобы узнать, например, сколько товаров относится к первому типу («Мебель»), мы можем сразу выполнить запрос к таблице **`items`**, используя значение её поля **`t_id`** для определения того факта, что некий товар относится именно к первому типу. Заметьте, что промежуточная таблица (**`category`**) здесь совершенно не нужна.

Аналогично, если нам нужно выяснить, к какому типу относится товар «Кружка», мы используем значение поля **`t_id`** таблицы **`item`** и сразу же на его основе делаем запрос к таблице **`type`**, никак не используя промежуточную таблицу **`category`**.

Теперь рассмотрим неидентифицирующую связь. При её организации первичный ключ родительского отношения в процессе миграции в дочернее становится «просто внешним ключом» и не входит в состав первичного ключа. Соответствующая схема и пример с данными показаны на рисунках 2.3.m и 2.3.n.

В данном случае компьютеры и комнаты являются вполне самостоятельными независимыми сущностями (в отличие от случаев проведения курсов, которые не имеют смысла в отрыве от самих курсов{65}), и связь показывает лишь тот факт, что такие-то компьютеры расположены в таких-то комнатах.

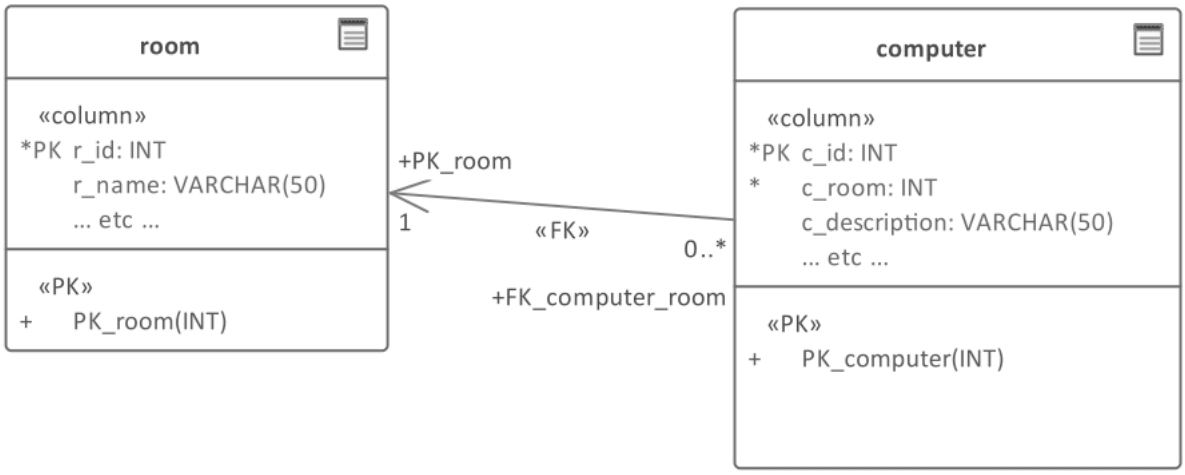


*Рисунок 2.3.m — **Схема, демонстрирующая неидентифицирующую связь***

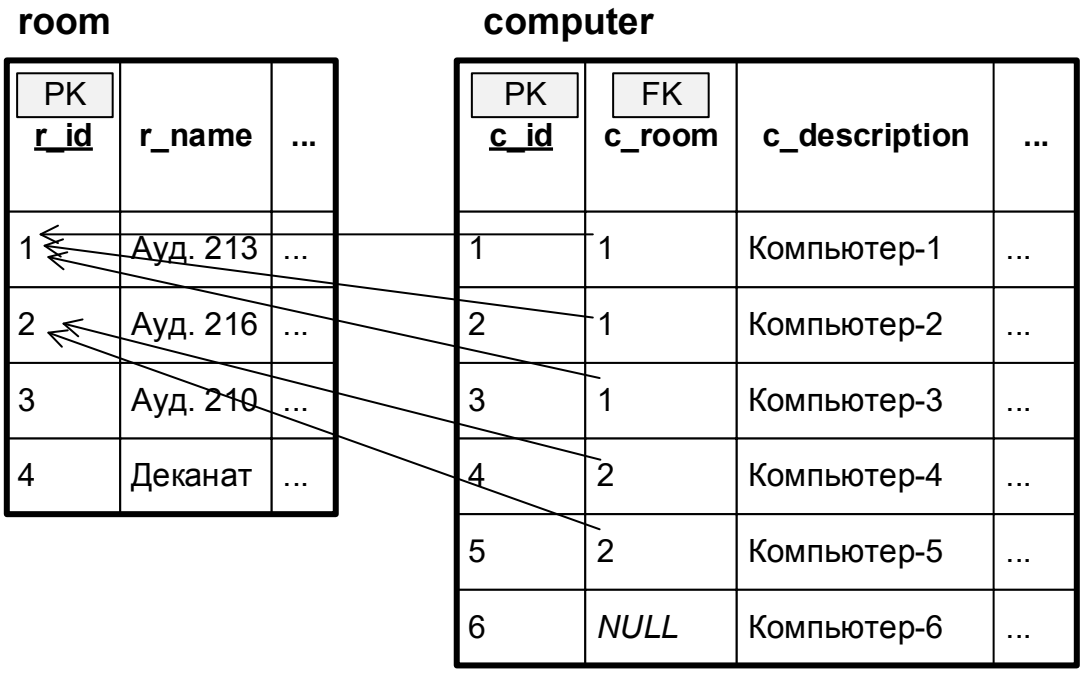


*Рисунок 2.3.n — **Пример данных, демонстрирующий неидентифицирующую связь***

Поскольку для создания неидентифицирующей связи нет необходимости внешний ключ дочернего отношения делать частью его же первичного ключа, мы получаем возможность безболезненно создавать в дочерних отношениях простые суррогатные первичные ключи даже в тех случаях, когда несколько отношений выстраиваются в цепочку связей (см. рисунок 2.3.o).

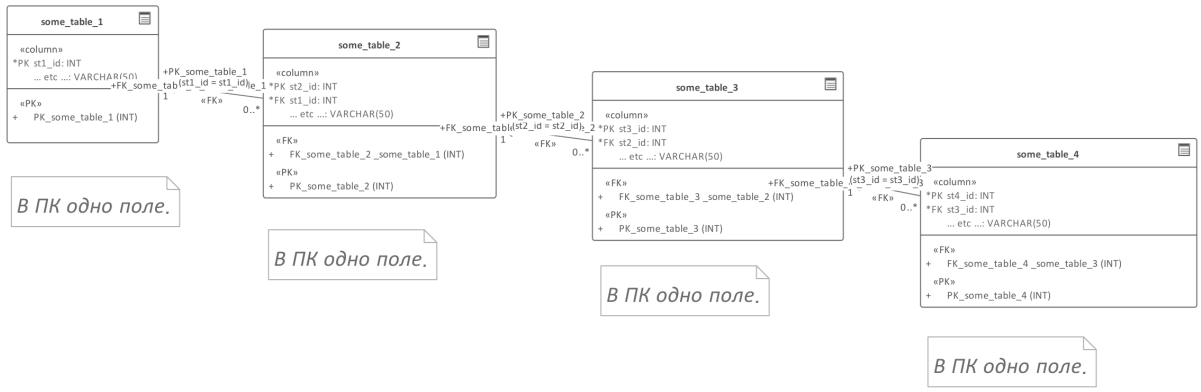


*Рисунок 2.3.о — **Отсутствие проблемы увеличения размера первичных ключей при использовании неидентифицирующих связей***

Легко догадаться, что положительная сторона характерного для идентифицирующих связей «накопления ключей» здесь становится отрицательной: мы можем «двигаться по цепи» только на один шаг в любую сторону. Если же между интересующими нас отношениями есть «промежуточное звено» (или несколько), мы обязаны использовать такие промежуточные отношения для получения интересующей нас информации. Рассмотрим это на примере (рисунок 2.3.р).

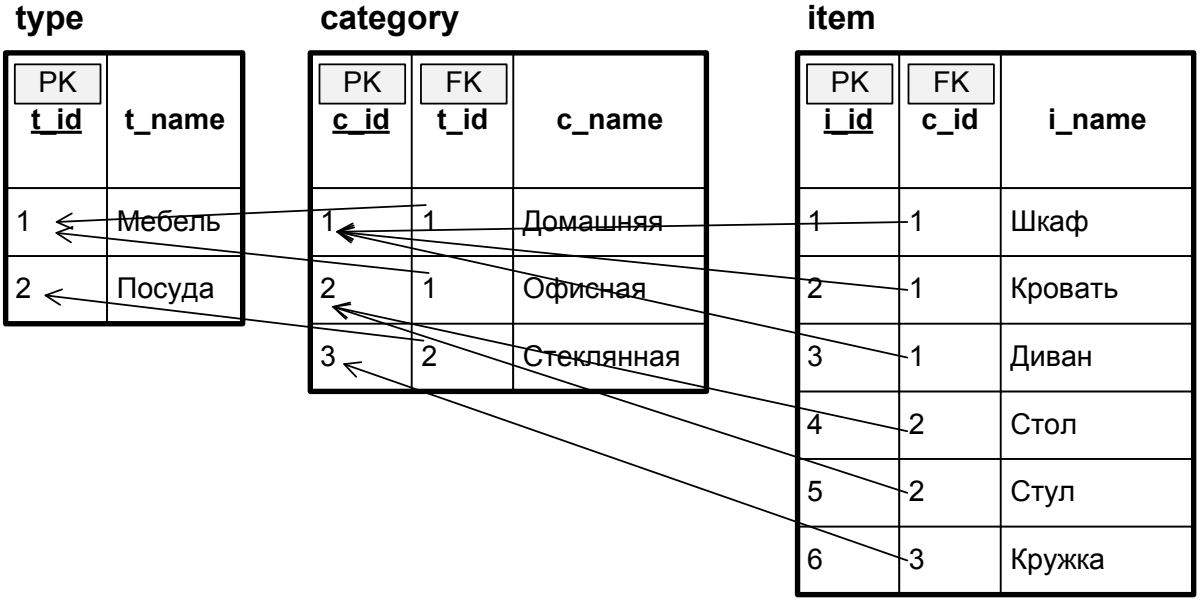


*Рисунок 2.3.р — **Пример данных для случая использования нескольких последовательных неидентифицирующих связей***

Чтобы узнать, например, сколько товаров относится к первому типу («Мебель»), мы не можем сразу выполнить запрос к таблице `items`, т.к. в ней нет информации о типах товаров (а есть информация только о категориях товаров). Мы должны сначала определить все категории товаров, относящиеся к интересующему нас типу, затем для каждой полученной категории определить количество относящихся к ней товаров, и лишь после этого, просуммировав полученные количества товаров, мы узнаем ответ на изначально поставленный вопрос.

Аналогично, если нам нужно выяснить, к какому типу относится товар «Кружка», мы должны сначала определить категорию этого товара, а затем (на основе информации о том, к какому типу товаров относится выясненная категория) мы уже можем получить информацию об интересующем нас типе товаров.

Ещё одним ключевым отличием идентифицирующей и неидентифицирующей связи является возможность существования во втором случае ситуации, когда запись в дочерней таблице «не привязана» ни к какой записи из родительской таблицы.

В случае идентифицирующей связи такая ситуация полностью исключена, т.к. использование **NULL**-значений в полях, входящих в состав первичного ключа, запрещено. Потому у любой записи в дочерней таблице во внешнем ключе (входящем в состав её первичного ключа) обязательно будет храниться некое конкретное значение первичного ключа родительской таблицы.

В случае неидентифицирующей связи такого ограничения нет, и мы можем хранить во внешнем ключе **NULL**-значения как признак того, что «родителя» у этой записи нет.

Доработаем пример, представленный на рисунке 2.3.n: представим, что компьютеры «Компьютер-2» и «Компьютер-5» пока не поставлены ни в какую комнату (например, пока просто хранятся на складе). Получается ситуация, представленная на рисунке 2.3.q:

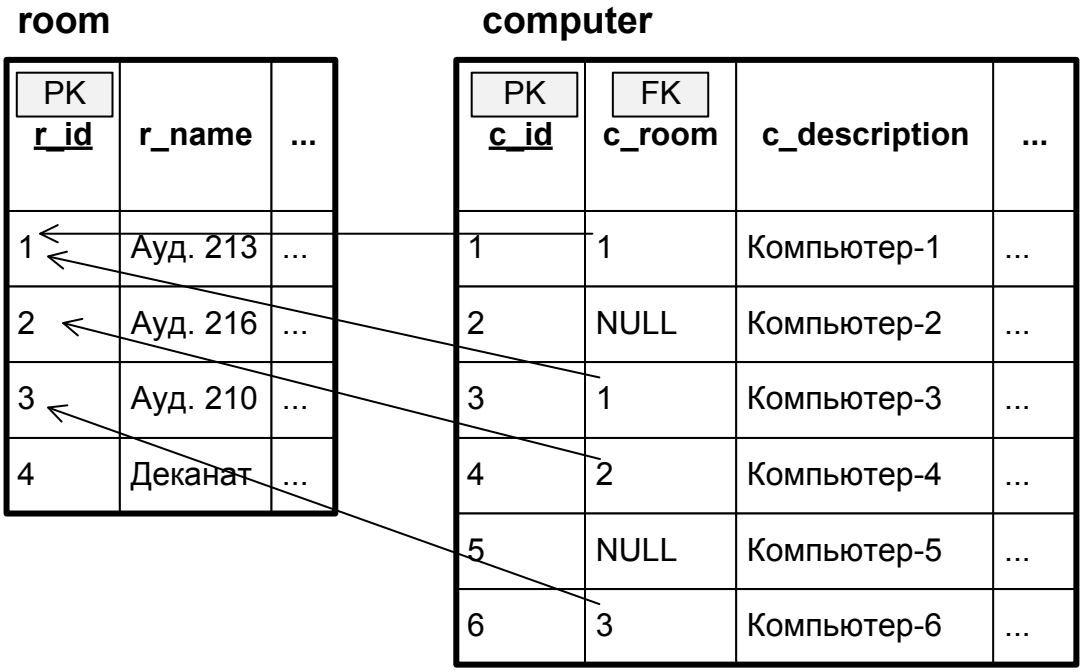


*Рисунок 2.3.q — **Пример использования NULL-значений при организации неидентифицирующей связи***

В некоторых случаях, напротив, нужно исключить возможность помещения в дочернюю таблицу записей без явного указания соответствующих им записей из родительской таблицы (допустим, по закону компьютеры, расположенные на складе, должны явно быть приписаны к комнате с названием «Склад»). Тогда на внешнем ключе дочерней таблицы активируется ограничение **NOT NULL**, предписывающее СУБД запрещать попытки вставки туда **NULL**-значений.

На практике именно создание неидентифицирующих связей с разрешением или запретом вставки **NULL**-значений во внешние ключи используется чаще всего: даже тогда, когда такое решение кажется сомнительным с точки зрения реляционной теории. Всё дело в удобстве управления данными и отсутствии дублирования полей (что на больших объёмах данных может приводить к ощутимому увеличению объёма базы данных).

Итого, если сравнить идентифицирующие и неидентифицирующие связи, получается следующая картина:

Идентифицирующая связь	Неидентифицирующая связь
Используется в случае, когда дочернее отношение является «неполноценным» без родительского.	Используется в случае, когда необходимо логически объединить самодостаточные отношения.
Первичный ключ родительского отношения становится внешним ключом дочернего отношения и входит в состав его первичного ключа.	Первичный ключ родительского отношения становится внешним ключом дочернего отношения, но **не** входит в состав его первичного ключа.
Запись в дочернем отношении не может существовать без соответствующей записи в родительском отношении.	Запись в дочернем отношении может существовать без соответствующей записи в родительском отношении. Если такая ситуация нежелательна (или недопустима), её можно исключить путём определения на внешнем ключе ограничения `NOT NULL`.
В случае цепи связей в каждом последующем дочернем отношении первичный ключ становится всё больше и больше.	В случае цепи связей в каждом последующем дочернем отношении первичный ключ не увеличивается.
В случае цепи связей для любого кортежа любого родительского отношения можно сразу получить список кортежей из любого дочернего отношения. А для любого кортежа любого дочернего отношения можно сразу получить информацию о родительском кортеже любого уровня.	В случае цепи связей для определения родительского элемента или списка дочерних элементов чаще всего придётся выполнять промежуточную операцию объединения.
В тех графических нотациях, которые поддерживают это отличие, изображается сплошной линией.	В тех графических нотациях, которые поддерживают это отличие, изображается пунктирной линией.

На этом мы заканчиваем теоретическую часть о связях как таковых, и сейчас рассмотрим ещё одну тему, которая заслуживает отдельной главы, т.к. не будет сильным преувеличением утверждение о том, что ради этой особенности во многом и создавались реляционные базы данных.

Задание 2.3.a: с помощью любого средства проектирования баз данных реализуйте по 5-10 фрагментов схем с использованием каждого связей «один ко многим» и «многие ко многим» и придумайте хотя бы один пример для случая связи «один к одному».

Задание 2.3.b: все ли связи в базе данных «Банк»{408} проведены корректно? Не пропущены ли там какие бы то ни было необходимые связи? Если вы считаете, что модель базы данных «Банк» нуждается в улучшении, внесите в неё соответствующие правки.

Задание 2.3.c: можно ли часть связей вида «многие ко многим» в базе данных «Банк»{408} заменить на связи «один ко многим»? Если вы считаете, что «да», внесите соответствующие правки в модель.

2.3.2. ССЫЛОЧНАЯ ЦЕЛОСТНОСТЬ И КОНСИСТЕНТНОСТЬ БАЗЫ ДАННЫХ

Ссылочная целостность (referential integrity[54]) — свойство реляционной базы данных, состоящее в неукоснительном соблюдении правила: если внешний ключ дочернего отношения содержит некоторое значение, это значение обязательно должно присутствовать в первичном ключе родительского отношения.

Упрощённо: запись в дочерней таблице не может ссылаться на несуществующую запись родительской таблицы.

Консистентность базы данных (database consistency[55]) — свойство реляционной базы данных, состоящее в неукоснительном соблюдении в любой момент времени всех ограничений, заданных неявно реляционной моделью или явно конкретной схемой базы данных.

Упрощённо: СУБД в любой момент времени контролирует типы данных, ссылочную целостность, ограничения типа **`CHECK`**{347}*, корректность выполнения триггеров*{350}*, корректность (валидность) запросов и т.д. и т.п.*

Начнём со ссылочной целостности.

Начиная с того момента, как между двумя отношениями явно проведена связь, СУБД при выполнении модификации данных в соответствующих таблицах проверят соблюдение указанного в определении правила: если внешний ключ дочернего отношения содержит некоторое значение, это значение обязательно должно присутствовать в первичном ключе родительского отношения.

Т.е. СУБД не позволит:

- добавить в дочернюю таблицу запись с таким значением внешнего ключа, которого нет среди значений первичного ключа родительской таблицы;
- добавить в дочернюю таблицу запись со значением внешнего ключа, равным **`NULL`**, если на этом внешнем ключе установлено ограничение **`NOT NULL`**;
- изменить у записи в дочерней таблице значение внешнего ключа на такое, которого нет среди значений первичного ключа родительской таблицы;
- изменить у записи в дочерней таблице значение внешнего ключа на **`NULL`**, если на этом внешнем ключе установлено ограничение **`NOT NULL`**;
- удалить из родительской таблицы запись, значение первичного ключа которой присутствует среди значений внешнего ключа дочерней таблицы (если включена соответствующая каскадная операция{73});
- изменить у записи в родительской таблице значение первичного ключа, если это значение присутствует среди значений внешнего ключа дочерней таблицы (если включена соответствующая каскадная операция{73});
- нарушить ограничение мощности связи, реализованное через триггеры[56].

Схематично изобразим основную идею на рисунке 2.3.r.

---

54 **Referential integrity** — the rule that no referencing tuple is allowed to exist if the corresponding referenced tuple doesn't also exist. («The New Relational Database Dictionary», C.J. Date)

55 **Database consistency** — … a database is in a state of consistency if and only if it conforms to all declared integrity constraints (database constraints and type constraints). («The New Relational Database Dictionary», C.J. Date)

56 «Работа с MySQL, MS SQL Server и Oracle в примерах» (С.С. Куликов), пример 33 [http://svyatoslav.biz/database_book/]

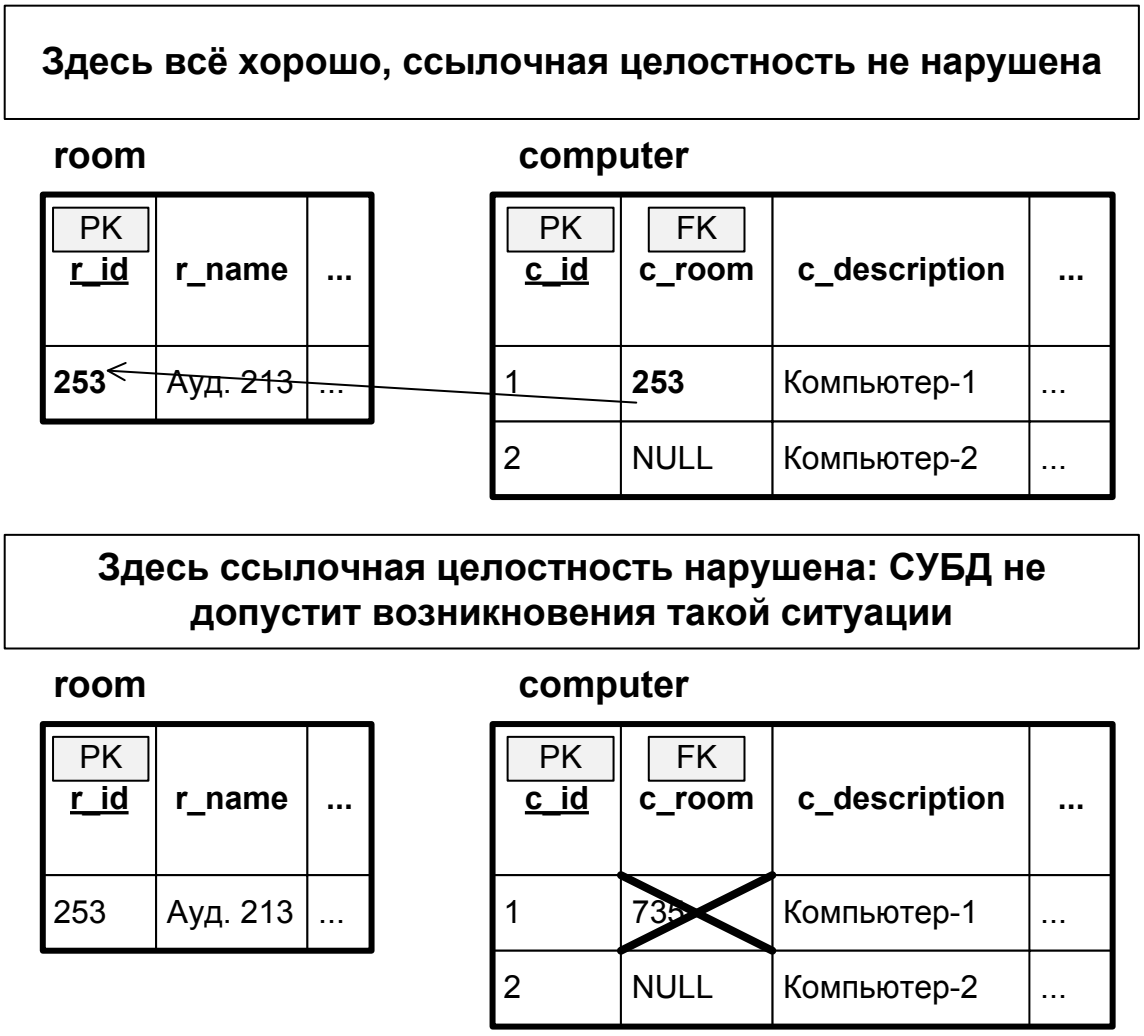


*Рисунок 2.3.r — **Основная идея ссылочной целостности***

Критически важно понимать, что СУБД не обладает «телепатией» и не может «догадаться» о том, какие значения надо сравнивать, если связи между таблицами не проведены **явно**.

Иногда в целях повышения совместимости с различными СУБД или из иных соображений связи между таблицами могут отсутствовать (хотя и нужны), а соответствующие проверки возлагаются на приложение, работающее с базой данных. Несмотря на то, что такое решение в некоторых редких случаях оправдано, оно является крайне рискованным.

Итак, пока мы рассмотрели случаи модификации данных в дочерней таблице — здесь всё достаточно просто, т.к. СУБД нужно всего лишь сравнивать значение внешнего ключа дочерней таблицы со значениями первичного ключа родительской таблицы. Пусть эта операция иногда может быть достаточно затратной, её алгоритм тривиален.

Но что происходит, если операция модификации выполняется с родительской таблицей? Тогда СУБД реализует ту или иную (указанную при организации связи) каскадную операцию.

Каскадные операции бывают следующих видов (представим их все сразу на одном рисунке для упрощения восприятия и запоминания — см. рисунок 2.3.s).

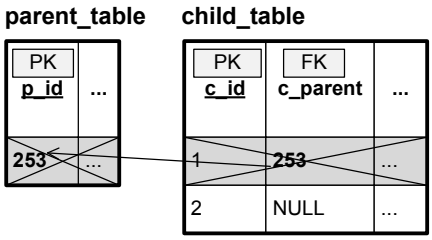
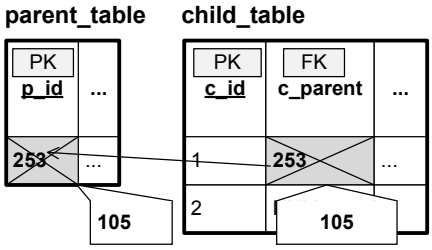
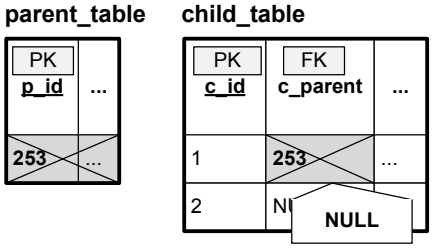
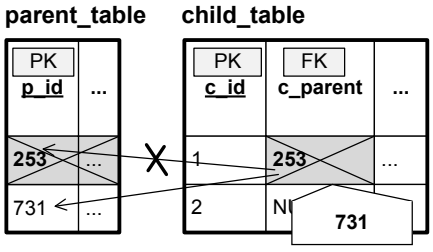
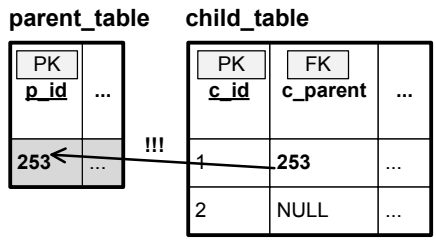
Название и суть операции	Схематичное пояснение	Пример предметной области
Каскадное удаление. При удалении записи из родительской таблицы автоматически удаляются все относящиеся к ней записи из дочерней таблицы.		На некотором новостном сайте принято при удалении рубрики новостей удалять и все относящиеся к ней новости.
Каскадное обновление. При изменении значения первичного ключа у записи в родительской таблице автоматически старое значение меняется на новое во внешнем ключе всех соответствующих записей дочерней таблицы.		Указание пользователя, выполнившего платёж, происходит по номеру паспорта. Если пользователь сменит паспорт, во всех его платежах старый номер придётся изменить на новый.
Установка пустых ключей. При удалении записи из родительской таблицы автоматически старое значение во внешнем ключе всех соответствующих записей дочерней таблицы меняется на `NULL`.		При расторжении договора аренды помещения (удаления информации об арендаторе) помещение становится свободным (арендатор не указан, т.е. `NULL`).
Установка значения по умолчанию. При удалении записи из родительской таблицы автоматически старое значение во внешнем ключе всех соответствующих записей дочерней таблицы меняется на некоторое новое (заданное заранее или вычисляемое прямо в процессе операции).		В случае увольнения сотрудника колл-центра все закреплённые за ним номера телефонов переносятся на руководителя подразделения.
Запрет каскадной операции. Бывает запрет каскадного удаления и запрет каскадного обновления. СУБД не позволит удалить из родительской таблицы запись (или изменить значение её первичного ключа), значение первичного ключа которой присутствует во внешнем ключе хотя бы одной записи дочерней таблицы.		В некотором интернет-магазине нельзя удалить категорию товаров, если к ней приписан хотя бы один товар.

*Рисунок 2.3.s — **Все виды каскадных операций***

Поскольку с каскадными операциями связано очень много непонимания, сразу рассмотрим несколько важных замечаний.

1. Любая каскадная операция активируется только при модификации данных в **родительской** таблице. Никогда никакие изменения данных в дочерней таблице не могут привести к запуску каскадной операции.

2. Любая каскадная операция активируется только при условии, что модификации данных в родительской таблице затрагивает **первичный ключ**. Первичный ключ гарантированно затрагивает только операция удаления записи (т.к. невозможно удалить «часть записи»). Если же выполняется изменение других полей, не входящих в состав первичного ключа, каскадные операции не активируются.

3. Вставка данных (в любую таблицу, не важно — родительскую или дочернюю) и выборка данных (также не важно, откуда) никогда не активируют никаких каскадных операций.

4. Не все СУБД умеют «своими силами» выполнять все виды каскадных операций. Если используемая вами СУБД не поддерживает ту или иную операцию, необходимой функциональности можно добиться с использованием триггеров{350}.

5. Почти все каскадные операции являются взаимоисключающими (кроме каскадного обновления, которое «совместимо» со всеми остальными операциями), а потому на некоторой одной связи не может быть включено несколько разных каскадных операций (например, одновременно **и** каскадное удаление, **и** установка значений по умолчанию)[57].

Очень часто можно услышать вопрос следующего вида: «А какую каскадную операцию тут правильно сделать?» (И при этом спрашивающий просто рисует схему из двух отношений.) Единственный честный и правильный ответ — любую. Выбор каскадной операции зависит **только** от предметной области.

Продемонстрируем это на примере новостного сайта, в котором есть рубрики новостей (родительская таблица) и сами новости (дочерняя таблица):

- если по правилам этого сайта при удалении рубрики нужно удалять все соответствующие новости, нужно включать каскадное удаление;
- если по правилам этого сайта при удалении рубрики нужно все соответствующие новости переносить в раздел «без рубрики», нужно включать установку пустых ключей;
- если по правилам этого сайта при удалении рубрики нужно все соответствующие новости переносить в некоторую заранее указанную рубрику (например, «Разное»), нужно включать установку значения по умолчанию;
- если по правилам этого сайта запрещено удалять рубрику, в которой есть хотя бы одна новость, нужно включать запрет каскадного удаления.

Иными словами: СУБД совершенно всё равно, что делать с вашими данными. Она в равной степени готова выполнить любую допустимую операцию. А вот какую именно операцию выполнять — решать вам как создателю базы данных.

---

57 Чисто теоретически можно создать такой триггер{352}, который будет на основе каких-то данных «выбирать», какую каскадную операцию выполнять в данный момент (и это будет выглядеть, как будто на одной связи реализовано несколько разных взаимоисключающих каскадных операций). Но без использования триггеров ни одна СУБД не позволяет реализовать такой вариант поведения.

В начале этой главы мы упомянули два определения: помимо ссылочной целостности СУБД также занимается обеспечением консистентности{72} базы данных.

Фактически, поддержание ссылочной целостности является одной из многих составляющих обеспечения консистентности базы данных (некоторые другие составляющие приведены в определении). Но поскольку данная глава посвящена связям и всем тем преимуществам, которые автоматически появляются при их создании, очень важно ещё раз подчеркнуть, что СУБД умеет контролировать только те ограничения, которые проистекают из самой сути реляционных баз данных (например, вы не сможете вставить данные в таблицу, указав неверные имена полей), и те, которые вы как создатель конкретной базы данных **явно** указали: связи, триггеры и т.д.

Что СУБД не умеет делать, так это «угадывать» ваши мысли. Поясним на примере очень частой ошибки — рассинхронизации данных.

Допустим, у нас есть база данных некоего новостного сайта, где очень много рубрик новостей, и в каждой рубрике очень много самих новостей. Для показа на сайте списка рубрик заказчик хочет в скобках выводить количество новостей в каждой рубрике и дату-время публикации самой свежей новости.

Существует два варианта решения поставленной задачи:

- каждый раз при формировании списка рубрик выполнять подсчёт количества новостей в каждой из них, а также определять дату-время самой свежей новости;
- завести в таблице, хранящей информацию о рубриках, два дополнительных поля (для хранения количества новостей в рубрике и даты-времени публикации самой свежей новости), и при формировании списка рубрик сразу брать готовые данные из этих полей — см. рисунок 2.3.t.

Первый вариант решения реализуется намного проще и не требует никаких доработок на уровне модели базы данных, но приводит к очень сильной потере производительности[58]. Второй вариант будет работать намного быстрее, но требует доработки базы данных.

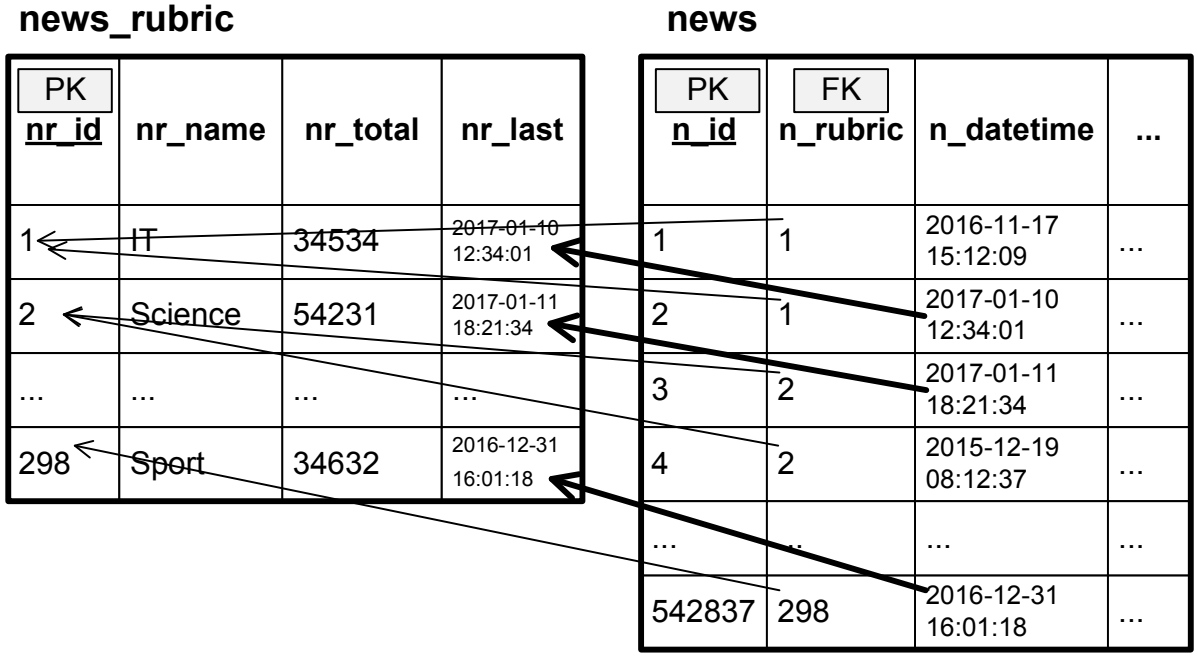


*Рисунок 2.3.t — **Фрагмент базы данных с агрегирующим и кэширующим полями***

58 Разница в производительности между первым и вторым вариантом может достигать миллинов раз и более.

Поле **nr_total** содержит в себе результаты подсчёта количества новостей по каждой рубрике — такие поля часто называют агрегирующими, т.е. содержащими в себе результаты некоей обработки (обобщения, агрегации) данных.

Поле **nr_last** содержит в себе информацию о дате-времени публикации самой свежей новости в каждой рубрике — такие поля часто называют кэширующими (в отличие от агрегирующих, они содержат просто копию данных, доступ к которым будет занимать намного меньше времени, чем в случае извлечения этих же данных из их основного места хранения).

Сложность этого варианта решения состоит в том, что СУБД без дополнительных доработок «не понимает», что значения полей **nr_total** и **nr_last** таблицы **news_rubric** как-то связаны с данными, хранящимися в таблице **news**. И потому ничто не мешает значениям этих полей оказаться некорректными (т.е. не соответствующими действительном количеству новостей в каждой рубрике и дате-времени публикации последней новости).

Чтобы исключить возможность возникновения такой рассинхронизации, необходимо на таблице **news** создать несколько триггеров{350}, которые будут автоматически выполняться при операциях вставки, обновления и удаления данных и обновлять соответствующие значения в таблице **news_rubric**.

Упрощённый пример подобной ситуации мы рассмотрим далее{101}. Также можно ознакомиться с подробной иллюстрацией решения схожей задачи в книге, посвящённой практическому использованию баз данных[59].

Итак, на этом с теорией всё — теперь рассмотрим связи на практике.

Задание 2.3.d: для всех схем, созданных вами при выполнении задания 2.3.a{71} исследуйте возможность и необходимость применения агрегирующих и кэширующих полей, внесите соответствующие правки в схемы.

Задание 2.3.e: как производительность базы данных «Банк»{408} можно повысить с использованием агрегирующих и кэширующих полей? Внесите соответствующие правки в модель.

Задание 2.3.f: какие каскадные операции необходимо установить на связях в базе данных «Банк»{408}? Внесите соответствующие правки в модель.

---

[59] «Работа с MySQL, MS SQL Server и Oracle в примерах» (С.С. Куликов), пример 32 [http://svyatoslav.biz/database_book/]

2.3.3. СОЗДАНИЕ И ИСПОЛЬЗОВАНИЕ СВЯЗЕЙ

Как было многократно сказано ранее, для организации связей используются первичный ключ{39} родительской таблицы и внешний ключ{47} дочерней таблицы, реализацию которых мы рассматривали в соответствующем разделе{50}.

Рассмотрение технической реализации связей «один ко многим» начнём с примера, представленного ранее на рисунках 2.3.a{58} и 2.3.b{58}.

Код для генерации соответствующего фрагмента модели базы данных выглядит следующим образом (приведём его для трёх разных СУБД).

MySQL **Пример кода к рисункам 2.3.a и 2.3.b**

```
CREATE TABLE `payment`
(
 `id` INT UNSIGNED NOT NULL AUTO_INCREMENT,
 `person` INT UNSIGNED NOT NULL,
 `money` DECIMAL(10,2) NOT NULL,
 CONSTRAINT `PK_payment` PRIMARY KEY (`id`)
);

CREATE TABLE `employee`
(
 `id` INT UNSIGNED NOT NULL AUTO_INCREMENT,
 `passport` CHAR(9) NOT NULL,
 `name` VARCHAR(50) NOT NULL,
 CONSTRAINT `PK_employee` PRIMARY KEY (`id`)
);

ALTER TABLE `employee`
 ADD CONSTRAINT `UQ_passport` UNIQUE (`passport`);

ALTER TABLE `payment` ADD CONSTRAINT `FK_payment_employee`
  FOREIGN KEY (`person`) REFERENCES `employee` (`id`)
   ON DELETE CASCADE ON UPDATE RESTRICT;
```

MS SQL **Пример кода к рисункам 2.3.a и 2.3.b**

```
CREATE TABLE [payment]
(
 [id] INT NOT NULL IDENTITY (1, 1),
 [person] INT NOT NULL,
 [money] MONEY NOT NULL
);

CREATE TABLE [employee]
(
 [id] INT NOT NULL IDENTITY (1, 1),
 [passport] CHAR(9) NOT NULL,
 [name] NVARCHAR(50) NOT NULL
);

ALTER TABLE [payment]
 ADD CONSTRAINT [PK_payment]
  PRIMARY KEY CLUSTERED ([id]);

ALTER TABLE [employee]
 ADD CONSTRAINT [PK_employee]
  PRIMARY KEY CLUSTERED ([id]);

ALTER TABLE [employee]
 ADD CONSTRAINT [UQ_passport] UNIQUE NONCLUSTERED ([passport]);

ALTER TABLE [payment] ADD CONSTRAINT [FK_payment_employee]
 FOREIGN KEY ([person]) REFERENCES [employee] ([id]) ON DELETE CASCADE;
```

Oracle Пример кода к рисункам 2.3.a и 2.3.b

```
CREATE TABLE "payment"
(
 "id" NUMBER(38) NOT NULL,
 "person" NUMBER(38) NOT NULL,
 "money" NUMBER(15,4) NOT NULL
);

CREATE TABLE "employee"
(
 "id" NUMBER(38) NOT NULL,
 "passport" CHAR(9) NOT NULL,
 "name" NVARCHAR2(50) NOT NULL
);

CREATE SEQUENCE "SEQ_payment_id"
 INCREMENT BY 1
 START WITH 1
 NOMAXVALUE
 MINVALUE 1;

CREATE OR REPLACE TRIGGER "TRG_payment_id"
 BEFORE INSERT
 ON "payment"
 FOR EACH ROW
 BEGIN
  SELECT "SEQ_payment_id".NEXTVAL
  INTO :NEW."id"
  FROM DUAL;
 END;

CREATE SEQUENCE "SEQ_employee_id"
 INCREMENT BY 1
 START WITH 1
 NOMAXVALUE
 MINVALUE 1;

CREATE OR REPLACE TRIGGER "TRG_employee_id"
 BEFORE INSERT
 ON "employee"
 FOR EACH ROW
 BEGIN
  SELECT "SEQ_employee_id".NEXTVAL
  INTO :NEW."id"
  FROM DUAL;
 END;

ALTER TABLE "payment"
 ADD CONSTRAINT "PK_payment"
  PRIMARY KEY ("id") USING INDEX;

ALTER TABLE "employee"
 ADD CONSTRAINT "PK_employee"
  PRIMARY KEY ("id") USING INDEX;

ALTER TABLE "employee"
 ADD CONSTRAINT "UQ_passport" UNIQUE ("passport") USING INDEX;

ALTER TABLE "payment"
 ADD CONSTRAINT "FK_payment_employee"
 FOREIGN KEY ("person") REFERENCES "employee" ("id") ON DELETE CASCADE;
```

В дальнейшем мы постараемся ограничиваться лишь примерами в синтаксисе MySQL (как наиболее компактном), но сейчас рассмотрим код подробнее, т.к. он содержит в себе части, в равной мере относящиеся к темам создания таблиц, ключей и связей.

Действие	MySQL	MS SQL Server	Oracle
Создание таблиц	Строки 1-7 и 9-15	Строки 1-6 и 8-13	Строки 1-6 и 8-13
Указание первичных ключей	Строки 3, 6, 11, 14 — объединено с созданием таблиц	Строки 15-17 и 19-21	Строки 47-49 и 51-53
Указание того факта, что первичные ключи являются автоинкрементируемыми	Строки 3 и 11 — объединено с созданием таблиц	Строки 3 и 10 — в MS SQL Server для достижения необходимого эффекта первичный ключ делают identity-полем	Строки 15-45 — в Oracle для достижения необходимого эффекта нужно создать последовательность (sequence) и использовать её значения с помощью триггера для формирования значения первичного ключа
Указание того факта, что значения поле «passport» должно быть уникальным	Строки 17-18	Строки 23-24	Строки 55-56
Создание связи «один ко многим»	Строки 20-22	Строки 26-27	Строки 58-60

Обратите внимание, что лишь в синтаксисе MySQL запрет каскадного обновления (свойство связи `ON UPDATE RESTRICT`) указано явно. У MS SQL Server такое поведения (называемое `NO ACTION`) включено по умолчанию и не требует явного указания[60], а Oracle вовсе не поддерживает каскадное обновление (но его можно эмулировать триггерами).

Теперь посмотрим, что нам дало наличие созданной связи.

Для начала наполним таблицу `employee` тестовыми данными:

id	passport	name
1	AA1231234	Иванов И.И.
2	BB1231234	Петров П.П.
3	CC1231234	Сидоров С.С.

MySQL **Пример кода для вставки тестовых данных в таблицу `employee`**

```
1   INSERT INTO `employee`
2               (`passport`, `name`)
3   VALUES      ('AA1231234', 'Иванов И.И.'),
4               ('BB1231234', 'Петров П.П.'),
5               ('CC1231234', 'Сидоров С.С.')
```

MS SQL **Пример кода для вставки тестовых данных в таблицу `employee`**

```
1   INSERT INTO [employee]
2               ([passport], [name])
3   VALUES      ('AA1231234', N'Иванов И.И.'),
4               ('BB1231234', N'Петров П.П.'),
5               ('CC1231234', N'Сидоров С.С.')
```

[60] Строго говоря, в MySQL по умолчанию также используется NO ACTION, но исторически принято писать RESTRICT явным образом.

Oracle Пример кода для вставки тестовых данных в таблицу `employee`

```
INSERT ALL
 INTO "employee" ("passport", "name") VALUES ('AA1231234', N'Иванов И.И.')
 INTO "employee" ("passport", "name") VALUES ('BB1231234', N'Петров П.П.')
 INTO "employee" ("passport", "name") VALUES ('CC1231234', N'Сидоров С.С.')
SELECT 1 FROM "DUAL"
```

Теперь наполним таблицу `payment` тестовыми данными:

id	person	money
1	1	100.00
2	2	400.00
3	1	150.00
4	2	800.00
5	2	900.00

MySQL Пример кода для вставки тестовых данных в таблицу `payment`

```
INSERT INTO `payment`
            (`person`, `money`)
VALUES      (1, 100),
            (2, 400),
            (1, 150),
            (2, 800),
            (2, 900)
```

MS SQL Пример кода для вставки тестовых данных в таблицу `payment`

```
INSERT INTO [payment]
            ([person], [money])
VALUES      (1, 100),
            (2, 400),
            (1, 150),
            (2, 800),
            (2, 900)
```

Oracle Пример кода для вставки тестовых данных в таблицу `payment`

```
INSERT ALL
 INTO "payment" ("person", "money") VALUES (1, 100)
 INTO "payment" ("person", "money") VALUES (2, 400)
 INTO "payment" ("person", "money") VALUES (1, 150)
 INTO "payment" ("person", "money") VALUES (2, 800)
 INTO "payment" ("person", "money") VALUES (2, 900)
SELECT 1 FROM "DUAL"
```

До сих пор все запросы выполнялись успешно. Но что будет, если мы попытаемся передать данные, нарушающие ссылочную целостность{72}? Проверим это, передав в поле `person` значение 4 (такого сотрудника нет) и значение `NULL` (его вставка тоже запрещена, т.к. у этого поля включено ограничение `NOT NULL`).

В приведённом ниже коде после каждого запроса в комментариях указано сообщение об ошибке, возвращённое СУБД в ответ на наши действия.

MySQL Попытка нарушения ссылочной целостности

```
INSERT INTO `payment`
            (`person`, `money`)
VALUES      (4, 999)
-- Error Code: 1452. Cannot add or update a child row: a foreign key
-- constraint fails

INSERT INTO `payment`
            (`person`, `money`)
VALUES      (NULL, 999)
-- Error Code: 1048. Column 'person' cannot be null
```

MS SQL Попытка нарушения ссылочной целостности

```
INSERT INTO [payment]
            ([person], [money])
VALUES      (4, 999)
-- Msg 547, Level 16, State 0, Line 1
-- The INSERT statement conflicted with the FOREIGN KEY constraint
-- "FK_payment_employee".
-- The statement has been terminated.

INSERT INTO [payment]
            ([person], [money])
VALUES      (NULL, 100)
-- Msg 515, Level 16, State 2, Line 1
-- Cannot insert the value NULL into column 'person', table 'payment';
-- column does not allow nulls. INSERT fails.
-- The statement has been terminated.
```

Oracle Попытка нарушения ссылочной целостности

```
INSERT INTO "payment"
            ("person", "money")
VALUES      (4, 999)
-- Error starting at line : 92 in command ...
-- SQL Error: ORA-02291: integrity constraint (FK_payment_employee)
-- violated - parent key not found
-- *Cause:    A foreign key value has no matching primary key value.
-- *Action:   Delete the foreign key or add a matching primary key.

INSERT INTO "payment"
            ("person", "money")
VALUES      (NULL, 999)
-- Error starting at line : 96 in command ...
-- SQL Error: ORA-01400: cannot insert NULL into ("payment"."person")
-- *Cause:    An attempt was made to insert NULL into previously
-- listed objects.
-- *Action:   These objects cannot accept NULL values.
```

Итак, все три СУБД успешно предотвратили попытку выполнения операций, противоречащих ограничениям модели базы данных.

Со вставкой мы разобрались. Попробуем теперь «поломать» базу данных обновлением и удалением данных.

Сначала попытаемся изменить значение первичного ключа у записи в родительской таблице с id=1 — эта операция должна быть запрещена (в свойствах связи указан запрет каскадного обновления).

Потом попробуем удалить запись с id=2 — эта операция должна выполниться успешно, и при этом из таблицы **`payment`** должны быть автоматически удалены все платежи сотрудника с id=2.

Для сравнения выполним те же операции с записью, у которой id=3 (у этого сотрудника нет ни одного платежа — а потому обновление его идентификатора должно пройти успешно, а удаление его записи не приведёт ни к каким сторонним изменениям в базе данных).

MySQL Демонстрация работы каскадных операций

```
UPDATE `employee`
SET    `id` = 7
WHERE  `id` = 1
-- Error Code: 1451. Cannot delete or update a parent row: a foreign key
-- constraint fails

DELETE FROM `employee`
WHERE  `id` = 2
-- 1 row(s) affected

UPDATE `employee`
SET    `id` = 9
WHERE  `id` = 3
-- 1 row(s) affected

DELETE FROM `employee`
WHERE  `id` = 9
-- 1 row(s) affected
```

MS SQL Демонстрация работы каскадных операций

```
UPDATE [employee]
SET    [id] = 7
WHERE  [id] = 1
-- Поскольку поле [id] является identity-полем, сообщение
-- об ошибке будет таким:
-- Msg 8102, Level 16, State 1, Line 1
-- Cannot update identity column 'id'.

-- Если свойство identity у поля [id] убрать, сообщение
-- об ошибке будет таким:
-- The UPDATE statement conflicted with the REFERENCE constraint
-- "FK_payment_employee".
-- The statement has been terminated.

DELETE FROM [employee]
WHERE  [id] = 2
-- (1 row(s) affected)

UPDATE [employee]
SET    [id] = 9
WHERE  [id] = 3
-- (1 row(s) affected)

DELETE FROM [employee]
WHERE  [id] = 9
-- (1 row(s) affected)
```

Oracle **Демонстрация работы каскадных операций**

```
1   UPDATE "employee"
2   SET    "id" = 7
3   WHERE  "id" = 1
4   -- Error starting at line : 108 in command ...
5   -- SQL Error: ORA-02292: integrity constraint (FK_payment_employee)
6   -- violated - child record found
7   -- *Cause: attempted to delete a parent key value that had a foreign
8               dependency.
9   -- *Action:   delete dependencies first then parent or disable constraint.
10
11  DELETE FROM "employee"
12  WHERE  "id" = 2
13  -- 1 rows deleted.
14
15  UPDATE "employee"
16  SET    "id" = 9
17  WHERE  "id" = 3
18  -- 1 rows updated.
19
20  DELETE FROM "employee"
21  WHERE  "id" = 9
22  -- 1 rows deleted.
```

Итак, все три СУБД ведут себя почти одинаково (и совершенно логично, в полном соответствии с теоретическими ожиданиями). Лишь в MS SQL Server существует особенность, связанная с тем, как в данной СУБД реализуются автоинкрементируемые первичные ключи — **`identity`**-поля запрещено обновлять, и потому, чтобы убедиться в срабатывании запрета каскадного обновления мы были вынуждены отключить соответствующее свойство у поля **`id`**. Отметим, что в реальности данная проблема почти никогда не возникает, т.к. **`identity`**-поле является искусственным (не несёт никакого смысла для предметной области), и потому нет необходимости его обновлять.

После выполнения всех показанных выше операций часть данных была удалена, и в нашей базе данных осталось лишь:

Данные в таблице **`employee`**

id	passport	name
1	AA1231234	Иванов И.И.

Данные в таблице **`payment`**

id	person	money
1	1	100.00
3	1	150.00

Восстановим исходные наборы данных и посмотрим, как выполняются запросы на выборку в случае, когда интересующие нас значения находятся в разных таблицах.

Данные в таблице **`employee`**

id	passport	name
1	AA1231234	Иванов И.И.
2	BB1231234	Петров П.П.
3	CC1231234	Сидоров С.С.

Данные в таблице **`payment`**

id	person	money
1	1	100.00
2	2	400.00
3	1	150.00
4	2	800.00
5	2	900.00

В такой ситуации чаще всего используются т.н. запросы на объединение (**JOIN**) или запросы с подзапросами. Продемонстрируем оба варианта[61].

Допустим, мы хотим получить список сотрудников и сумму платежей по каждому из них. Ожидаемый результат будет выглядеть так:

name	sum
Иванов И.И.	250.00
Петров П.П.	2100.00
Сидоров С.С.	NULL

Значение **NULL** для Сидорова (к которому не относится ни одного платежа) здесь совершенно корректно и даже более выгодно, чем «0» (хоть и ноль получить несложно) — так мы подчёркиваем тот факт, что платежей не было, а не что их сумма равна нулю.

Итак, код SQL-запросов для получения этого результата будет выглядеть следующим образом:

MySQL **Пример SQL-запроса на объединение (JOIN) к двум таблицам**

```
SELECT `name`,
       SUM(`money`) AS `sum`
FROM   `employee`
       LEFT OUTER JOIN `payment`
                    ON `employee`.`id` = `payment`.`person`
GROUP  BY `employee`.`id`
```

MS SQL **Пример SQL-запроса на объединение (JOIN) к двум таблицам**

```
SELECT [name],
       SUM([money]) AS [sum]
FROM   [employee]
       LEFT OUTER JOIN [payment]
                    ON [employee].[id] = [payment].[person]
GROUP  BY [employee].[id],
          [employee].[name]
```

Oracle **Пример SQL-запроса на объединение (JOIN) к двум таблицам**

```
SELECT "name",
       SUM("money") AS "sum"
FROM   "employee"
       LEFT OUTER JOIN "payment"
                    ON "employee"."id" = "payment"."person"
GROUP  BY "employee"."id",
          "employee"."name"
```

На рисунке 2.3.u схематично показана логика работы СУБД при выполнении подобных запросов.

Если расписать её словами, получается:

- СУБД берёт все записи таблицы **employee** и ищет им соответствия из таблицы **payment** по признаку равенства значений полей **id** (из таблицы **employee**) и **person** (из таблицы **payment**);
- если совпадения для какой-то из записей таблицы **employee** не обнаружено, вместо данных из таблицы **payment** подставляются **NULL**-значения;
- получив набор сопоставленных из двух таблиц данных (на рисунке 2.3.u соответствующая информация представлена в блоке «Результат выполнения LEFT OUTER JOIN»), СУБД выполняет суммирование значений платежей в рамках групп записей с одинаковым значением поля **id**;

[61] Рассмотрение языка SQL выходит за рамки данной книги, но вы можете ознакомиться с огромным количеством подробно пояснённых примеров в книге «Работа с MySQL, MS SQL Server и Oracle в примерах» (С.С. Куликов) [http://svyatoslav.biz/database_book/]

- после выполнения этой операции мы получаем итоговый результат (на рисунке 2.3.и соответствующая информация представлена в блоке «Результат выполнения SUM совместно с GROUP BY»).

Исходные данные

employee

PK id	passport	name
1	AA1231234	Иванов И.И.
2	BB1231234	Петров П.П.
3	CC1231234	Сидоров С.С.

payment

PK id	FK person	money
1	1	100
2	2	400
3	1	150
4	2	800
5	2	900

Результат выполнения LEFT OUTER JOIN

id	person	name	money
1	1	Иванов И.И.	100
2	2	Петров П.П.	400
1	1	Иванов И.И.	150
2	2	Петров П.П.	800
2	2	Петров П.П.	900
3	NULL	Сидоров С.С.	NULL

Результат выполнения SUM совместно с GROUP BY

id	person	name	money
1	1	Иванов И.И.	250
2	2	Петров П.П.	2100
3	NULL	Сидоров С.С.	NULL

*Рисунок 2.3.и — **Упрощённая логика запросов к двум таблицам***

Теперь рассмотрим ещё один пример, основанный на том же наборе данных. Допустим, мы хотим получить список сотрудников, у которых не было ни одного платежа. Эта задача может быть решена как с использованием запроса на объединение (JOIN), так и с использованием подзапроса.

Ожидаемый результат будет выглядеть так:

name
Сидоров С.С.

MySQL Пример решения задачи с использованием запроса на объединение (`JOIN`) и подзапроса

```
1   -- Вариант 1 (с использованием JOIN):
2   SELECT `name`
3   FROM   `employee`
4          LEFT OUTER JOIN `payment`
5                       ON `employee`.`id` = `payment`.`person`
6   WHERE  `payment`.`person` IS NULL
7
8   -- Вариант 2 (с использованием подзапроса):
9   SELECT `name`
10  FROM   `employee`
11  WHERE  `id` NOT IN (SELECT `person`
12                      FROM   `payment`)
```

MS SQL Пример решения задачи с использованием запроса на объединение (`JOIN`) и подзапроса

```
1   -- Вариант 1 (с использованием JOIN):
2   SELECT [name]
3   FROM   [employee]
4          LEFT OUTER JOIN [payment]
5                       ON [employee].[id] = [payment].[person]
6   WHERE  [payment].[person] IS NULL
7
8   -- Вариант 2 (с использованием подзапроса):
9   SELECT [name]
10  FROM   [employee]
11  WHERE  [id] NOT IN (SELECT [person]
12                      FROM   [payment])
```

Oracle Пример решения задачи с использованием запроса на объединение (`JOIN`) и подзапроса

```
1   -- Вариант 1 (с использованием JOIN):
2   SELECT "name"
3   FROM   "employee"
4          LEFT OUTER JOIN "payment"
5                       ON "employee"."id" = "payment"."person"
6   WHERE  "payment"."person" IS NULL
7
8   -- Вариант 2 (с использованием подзапроса):
9   SELECT "name"
10  FROM   "employee"
11  WHERE  "id" NOT IN (SELECT "person"
12                      FROM   "payment")
```

Решения для всех трёх СУБД получаются совершенно одинаковыми и работают следующим образом.

Вариант 1:

- выполняя **`LEFT OUTER JOIN`**, СУБД получает тот же набор данных, что был рассмотрен в предыдущем примере (на рисунке 2.3.u соответствующая информация представлена в блоке «Результат выполнения LEFT OUTER JOIN»);
- в отличие от предыдущего примера, где после этой операции выполнялось суммирование по группам записей, в данном случае СУБД нужно лишь отобрать записи, не нашедшие себе «пары» из таблицы **`payment`** — признаком такой ситуации является значение **`NULL`** поля **`person`**.

Вариант 2:

- в первую очередь СУБД выполнит подзапрос (представлен в строках 11-12 решений для всех трёх СУБД) и получит список идентификаторов сотрудников, у которых были платежи;
- теперь СУБД будет выбирать из таблицы **`employee`** те записи, идентификаторы (**`id`**) которых **не** входят в набор, полученный на предыдущем шаге.

Представленные варианты решения отличаются не только синтаксически. Существует множество особенностей поведения разных СУБД в зависимости от имеющихся наборов данных и нюансов самих запросов (к сожалению, рассмотрение этих деталей выходит за рамки данной книги). Также эти варианты могут существенно различаться по производительности (в общем случае вариант с **JOIN** должен работать быстрее, но могут быть и исключения).

На этом мы завершаем рассмотрение связей «один ко многим» и переходим к связям «многие ко многим». Рассмотрение их технической реализации начнём с примера, представленного ранее на рисунках 2.3.e{61} и 2.3.f{61}.

Код для генерации соответствующего фрагмента модели базы данных выглядит следующим образом. Как и было обещано ранее, теперь мы будем рассматривать код только для MySQL.

MySQL **Пример кода к рисункам 2.3.e и 2.3.f**

```
CREATE TABLE `m2m_student_subject`
(
 `st_id` INT NOT NULL,
 `sb_id` INT NOT NULL,
 CONSTRAINT `PK_m2m_student_subject` PRIMARY KEY (`st_id`,`sb_id`)
);

CREATE TABLE `subject`
(
 `sb_id` INT NOT NULL AUTO_INCREMENT,
 `sb_name` VARCHAR(50) NOT NULL,
 CONSTRAINT `PK_subject` PRIMARY KEY (`sb_id`)
);

CREATE TABLE `student`
(
 `st_id` INT NOT NULL AUTO_INCREMENT,
 `st_name` VARCHAR(50) NOT NULL,
 CONSTRAINT `PK_student` PRIMARY KEY (`st_id`)
);

ALTER TABLE `m2m_student_subject`
 ADD CONSTRAINT `FK_m2m_student_subject_student`
  FOREIGN KEY (`st_id`) REFERENCES `student` (`st_id`)
   ON DELETE CASCADE ON UPDATE CASCADE;

ALTER TABLE `m2m_student_subject`
 ADD CONSTRAINT `FK_m2m_student_subject_subject`
  FOREIGN KEY (`sb_id`) REFERENCES `subject` (`sb_id`)
   ON DELETE CASCADE ON UPDATE CASCADE;
```

В данном коде нет ничего принципиально нового по сравнению с рассмотренным ранее (в примере, посвящённом связям «один ко многим»). Разве что стоит ещё раз подчеркнуть: связи «многие ко многим» как таковой на уровне базы данных не существует — она формируется из двух связей «один ко многим», проведённых от связываемых таблиц (в нашем случае — **student** и **subject**) к т.н. «таблице связи» (в нашем случае — **m2m_student_subject**).

Наполним таблицу **student** тестовыми данными:

st_id	st_name
1	Иванов И.И.
2	Петров П.П.
3	Сидоров С.С.

Наполним таблицу **subject** тестовыми данными:

sb_id	sb_name
1	Математика
2	Физика
3	Химия

Наполним таблицу **m2m_student_subject** тестовыми данными:

st_id	sb_id
1	2
2	3
3	3

MySQL Пример кода для вставки тестовых данных в таблицу **student**

```
INSERT INTO `student`
            (`st_name`)
VALUES      ('Иванов И.И.'),
            ('Петров П.П.'),
            ('Сидоров С.С.')
```

MySQL Пример кода для вставки тестовых данных в таблицу **subject**

```
INSERT INTO `subject`
            (`sb_name`)
VALUES      ('Математика'),
            ('Физика'),
            ('Химия')
```

MySQL Пример кода для вставки тестовых данных в таблицу **m2m_student_subject**

```
INSERT INTO `m2m_student_subject`
            (`st_id`, `sb_id`)
VALUES      (1, 2),
            (1, 3),
            (3, 3)
```

Ранее{60} было обещано пояснить, почему в таблице **m2m_student_subject** оба поля (**st_id** и **sb_id**), являющиеся внешними ключами, одновременно входят в состав составного естественного первичного ключа.

Поясним этот момент на примере решения следующей задачи. Допустим, нам нужно выяснить, сколько студентов изучает каждый предмет. Ожидаемый результат выглядит следующим образом.

sb_name	attendees
Физика	1
Химия	2

Для получения результата в данном случае достаточно использовать только две таблицы (логика построения запроса аналогична рассмотренной в примерах, посвящённых связям «один ко многим»).

MySQL Пример кода для решения поставленной задачи

```
SELECT `sb_name`,
       COUNT(`st_id`) AS `attendees`
FROM   `subject`
       JOIN `m2m_student_subject` USING(`sb_id`)
GROUP  BY `sb_id`
```

Сначала СУБД находит все пары записей из таблиц **subject** и **m2m_student_subject** (по признаку равенства значения полей **sb_id**), затем группирует полученный результат по значению поля **sb_id** и подсчитывает количество записей в каждой такой группе. Так и получается итоговый результат.

Теперь вернёмся к составному первичному ключу таблицы **m2m_student_subject** и представим, что этого ограничения нет, т.е. поля **st_id** и **sb_id** не входят в состав составного первичного ключа, и совокупность их значений не обязана быть уникальной. Тогда ничто не мешает нам поместить в таблицу **m2m_student_subject**, например, следующие данные:

st_id	sb_id
1	2
1	3
3	3
3	3
3	3
3	3

Выполнив только что рассмотренный SQL-запрос на таком наборе данных, мы получим:

sb_name	attendees
Физика	1
Химия	5

Т.е. химию изучает пять студентов. Но если посмотреть на данные в таблице **student**, легко заметить, что у нас всего три студента, т.е. никоим образом не может быть пятеро изучающих химию. Именно для исключения таких ситуаций (которые в реальной жизни могут иметь серьёзные экономические и юридические последствия) внешние ключи в «таблице связей» и вводят в состав естественного составного первичного ключа.

Иногда можно услышать возражение, состоящее в том, что в других предметных областях, напротив, необходимо многократно дублировать строки в таблице связей. Возможно. Но куда рациональнее завести в этой таблице дополнительное поле (своего рода счётчик) и хранить там «число повторений», а саму пару значений первичных ключей связываемых таблиц всё же хранить в таблице связи ровно один раз. Такое решение оказывается и более производительным, и более соответствующим концепции реляционных баз данных.

Продолжим данный пример, усложнив задачу: кроме количества студентов, изучающих каждый предмет, нужно также получить список их имён. Т.е. результат должен принять такой вид.

sb_name	attendees	names
Физика	1	Иванов И.И.
Химия	2	Иванов И.И.,Сидоров С.С.

Код для получения этого результата выглядит следующим образом.

MySQL **Пример кода для решения поставленной задачи**

```
SELECT `sb_name`,
       COUNT(`st_id`) AS `attendees`,
       GROUP_CONCAT(`st_name`) AS `names`
FROM   `subject`
       JOIN `m2m_student_subject` USING(`sb_id`)
       JOIN `student` USING(`st_id`)
GROUP  BY `sb_id`
```

Здесь мы уже не можем обойтись запросом к только лишь двум таблицам, т.к. имена студентов, названия предметов и информация о том, кто из студентов изучает какой предмет, хранится в трёх разных таблицах. Общая упрощённая логика выполнения подобных запросов представлена на рисунке 2.3.v.

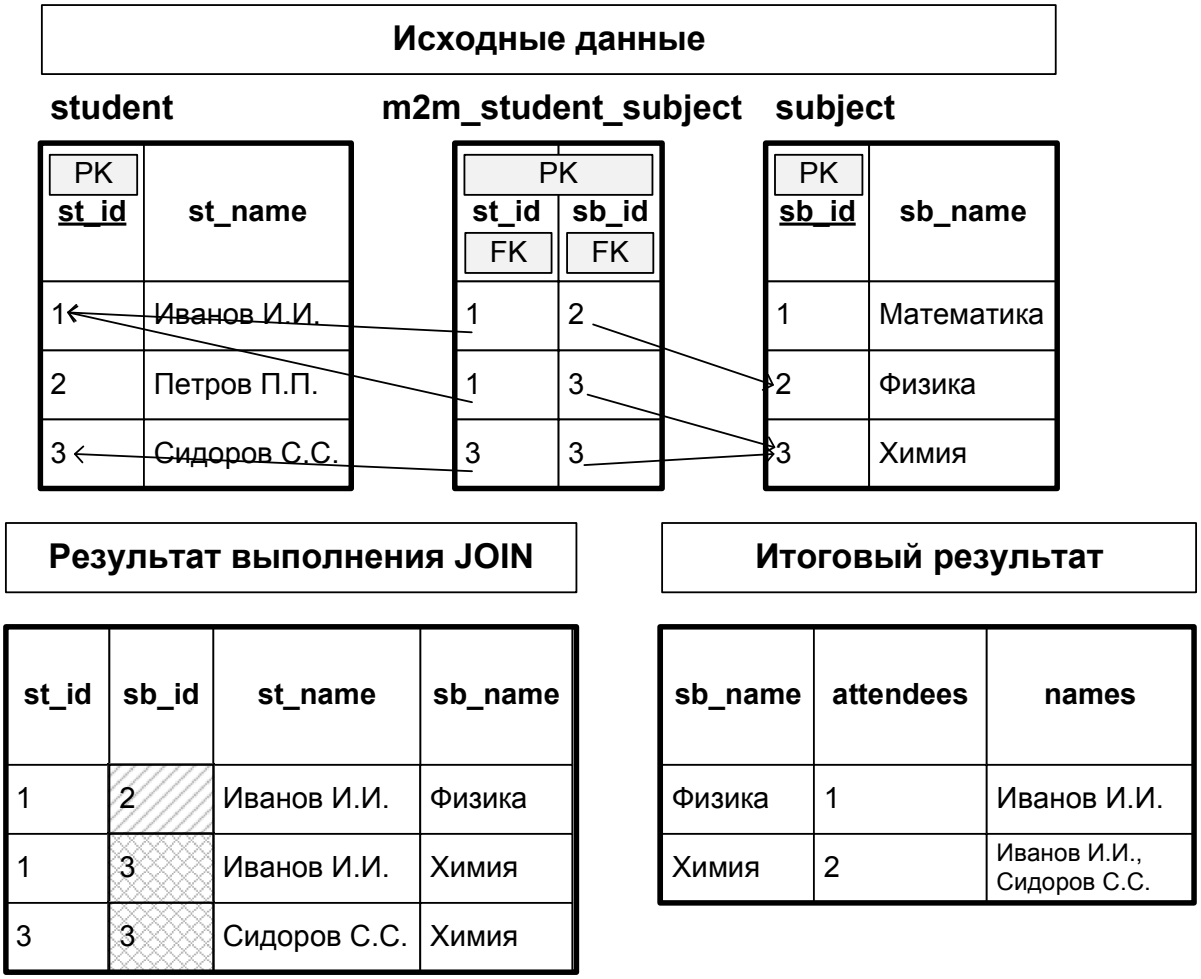


*Рисунок 2.3.v — **Упрощённая логика запросов к трём таблицам***

Здесь мы совершенно осознанно не будем рассматривать внутреннюю логику работы различных видов `JOIN`, группировок и т.д. (всё это есть в книге, посвящённой практическому использованию баз данных[62]) — сейчас важно понимать общий принцип того, как используются связи тех или иных видов, т.к. этот принцип остаётся неизменным в любой реляционной СУБД, а вот конкретные способы его реализации исчисляются десятками.

Теперь рассмотрим самый сложный и нетипичный пример, который будет иллюстрировать использование связей «один к одному» и триггеров{350}, обеспечивающих консистентность данных.

Представим, что мы создаём базу данных интернет-магазина, торгующего компьютерными комплектующими. Очевидно, что у товаров в таком магазине есть как общий свойства (например, название и цена), так и уникальные свойства, зависящие от вида товара (например, у монитора есть длина диагонали экрана, а у процессора такого свойства нет, зато есть другие, отсутствующие у мониторов).

62 «Работа с MySQL, MS SQL Server и Oracle в примерах» (С.С. Куликов) [http://svyatoslav.biz/database_book/]

Если мы попытаемся разместить в одной таблице информацию обо всех товарах, мы получим два крайне неприятных эффекта:

- такая таблица должна будет содержать тысячи полей (т.к. придётся перечислить все возможные свойства всех видов товаров) — и уже одно это делает подобное решение невозможным, т.к. у любой СУБД есть ограничение на количество полей в таблице, и оно, как правило, измеряется десятками;
- даже если бы нам удалось создать такую огромную таблицу, она почти полностью была бы заполнена `NULL`-значениями (т.к. если к некоторому виду товаров неприменимо некоторое свойство, соответствующее значение стоит выставить в `NULL`) — т.е. хранение данных было бы выполнено неоптимально.

Альтернативным (и очень хорошо зарекомендовавшим себя) решением является использование частного случая[63] схемы типа «звезда» с набором связей «один к одному».

Чтобы пример был проще для понимания, сознательно ограничим количество категорий товаров двумя («Принтеры», «Процессоры»), а также для каждой категории товаров приведём лишь по паре характеристик.

Получившаяся схема представлена на рисунке 2.3.w.

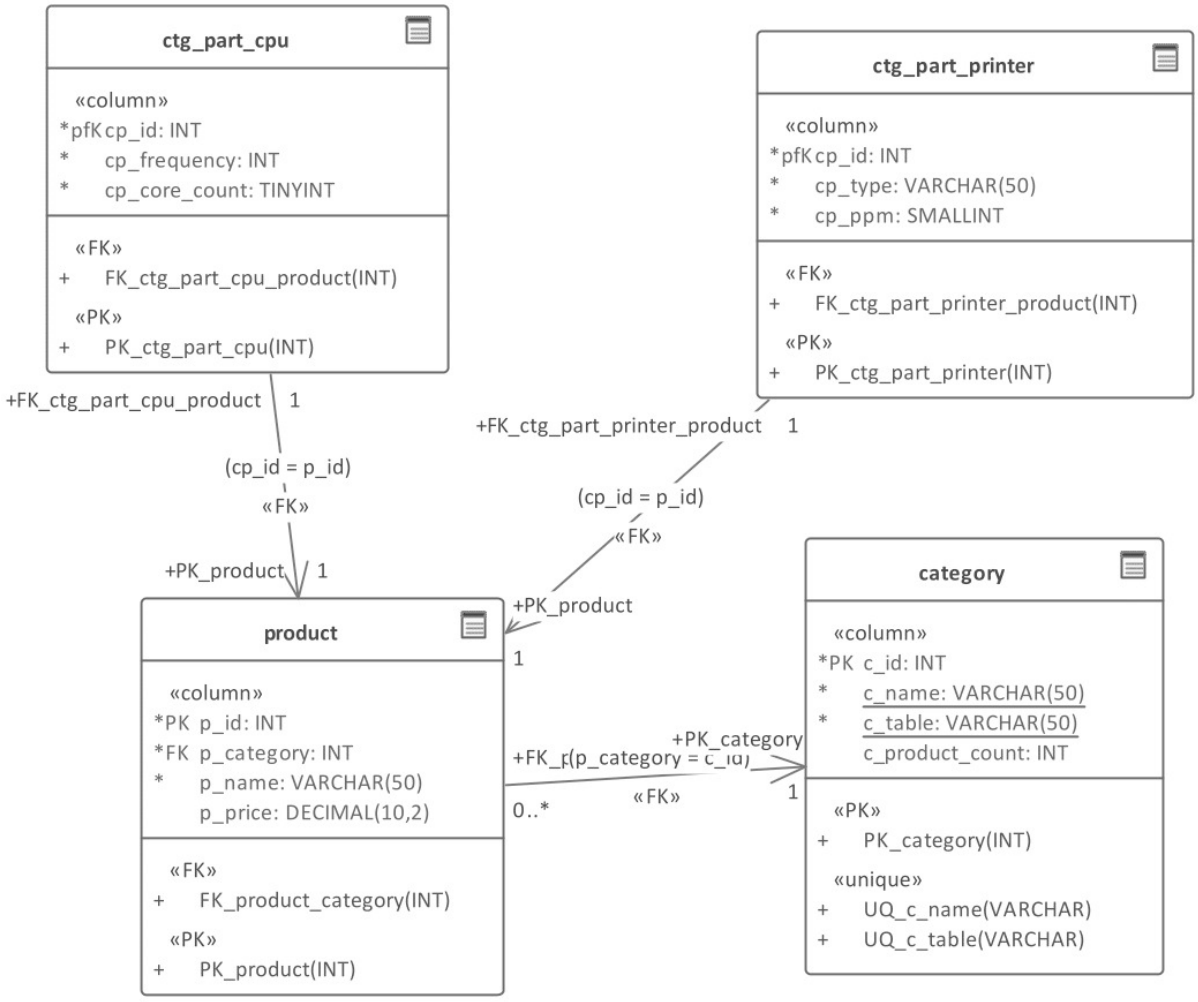


*Рисунок 2.3.w — **Схема, иллюстрирующая использование связей «один к одному»***

63 Это именно частный случай, т.к. в классической схеме «звезда» внешние ключи находятся в «центральной» таблице, а в данном частном случае — в дочерних таблицах.

Итак, связи «один к одному» здесь установлены между родительской таблицей **product** и её дочерними таблицами **ctg_part_cpu** и **ctg_part_printer** (и будут проведены к любой новой дочерней таблице, описывающей товары определённой категории).

Мощность связи («1 … 1») обеспечивается за счёт того, что внешние ключи в дочерних таблицах одновременно являются и первичными. Также особенностью представленного решения являются два достаточно строгих правила:

- первичные ключи всех дочерних таблиц, участвующих в связях «один к одному», должны называться одинаково (в нашем случае — **cp_id**), т.к. в противном случае построение запросов в разы усложнится;
- имена всех дочерних таблиц, участвующих в связях «один к одному», должны начинаться с префикса **ctg_part_**, т.к. в противном случае усложнится написание кода приложений, работающих с такой базой данных (по этому поводу см. пояснение в следующем абзаце).

Предложенное решение является достаточно оптимальным с точки зрения хранения данных, но оно влечёт за собой один очень серьёзный риск.

Сначала отметим, что существует достаточно строгое правило: приложение, работающее с базой данных, может изменять любые данные, но ни при каких обстоятельствах не должно изменять структуру базы данных (как минимум, потому, что с ней может работать много других приложений, не ожидающих таких изменений; и сама процедура внесения изменений в структуру базы данных может привести к необратимым повреждениям в ней).

Но что делать в нашей ситуации, если нужно добавить новую категорию товаров, удалить или отредактировать (изменить набор характеристик товара) существующую? У нас нет другого выхода, кроме как изменить структуру базы данных (добавить новую таблицу и связь, удалить таблицу, изменить набор полей в таблице).

Именно поэтому выше были обозначены «два строгих правила» (в контексте обсуждаемого риска особенно актуально второе) — они позволяют пусть и не исключить, но минимизировать вероятность возникновения проблем со структурой базы данных.

Создадим на основе представленной схемы базу данных, наполним её тестовыми данными и исследуем логику работы различных операций.

Данные будут такими.

В таблице **category**:

c_id	c_name	c_table	c_product_count
1	Принтеры	ctg_part_printer	3
2	Процессоры	ctg_part_cpu	2

В таблице **product**:

p_id	p_category	p_name	p_price
1	1	Дешёвый принтер	100.00
2	1	Дорогой принтер	1000.00
3	1	Просто принтер	300.00
4	2	Дешёвый процессор	500.00
5	2	Дорогой процессор	7000.00

В таблице `ctg_part_printer`:

cp_id	cp_type	cp_ppm
1	laser	10
2	laser	30
3	ink-jet	5

В таблице `ctg_part_cpu`:

cp_id	cp_frequency	cp_core_count
4	1800	2
5	3500	6

Код выглядит следующим образом.

MySQL **Пример кода к рисунку 2.3.w (создание таблиц и связей)**

```
CREATE TABLE `product`
(
 `p_id` INT NOT NULL AUTO_INCREMENT,
 `p_category` INT NOT NULL,
 `p_name` VARCHAR(50) NOT NULL,
 `p_price` DECIMAL(10,2)  NULL,
 CONSTRAINT `PK_product` PRIMARY KEY (`p_id`)
);

CREATE TABLE `ctg_part_printer`
(
 `cp_id` INT NOT NULL,
 `cp_type` VARCHAR(50) NOT NULL,
 `cp_ppm` SMALLINT NOT NULL,
 CONSTRAINT `PK_ctg_part_printer` PRIMARY KEY (`cp_id`)
);

CREATE TABLE `ctg_part_cpu`
(
 `cp_id` INT NOT NULL,
 `cp_frequency` INT NOT NULL,
 `cp_core_count` TINYINT NOT NULL,
 CONSTRAINT `PK_ctg_part_cpu` PRIMARY KEY (`cp_id`)
);

CREATE TABLE `category`
(
 `c_id` INT NOT NULL AUTO_INCREMENT,
 `c_name` VARCHAR(50) NOT NULL,
 `c_table` VARCHAR(50) NOT NULL,
 `c_product_count` INT    NULL,
 CONSTRAINT `PK_category` PRIMARY KEY (`c_id`)
);

ALTER TABLE `category`
 ADD CONSTRAINT `UQ_c_name` UNIQUE (`c_name`);

ALTER TABLE `category`
 ADD CONSTRAINT `UQ_c_table` UNIQUE (`c_table`);

ALTER TABLE `product`
 ADD CONSTRAINT `FK_product_category`
  FOREIGN KEY (`p_category`) REFERENCES `category` (`c_id`)
   ON DELETE CASCADE ON UPDATE CASCADE;

```

```
ALTER TABLE `ctg_part_printer`
 ADD CONSTRAINT `FK_ctg_part_printer_product`
  FOREIGN KEY (`cp_id`) REFERENCES `product` (`p_id`)
   ON DELETE CASCADE ON UPDATE CASCADE;

ALTER TABLE `ctg_part_cpu`
 ADD CONSTRAINT `FK_ctg_part_cpu_product`
  FOREIGN KEY (`cp_id`) REFERENCES `product` (`p_id`)
   ON DELETE CASCADE ON UPDATE CASCADE;
```

MySQL Пример кода к рисунку 2.3.w (вставка данных)

```
INSERT INTO `category`
            (`c_name`, `c_table`, `c_product_count`)
VALUES      ('Принтеры', 'ctg_part_printer', 3),
            ('Процессоры', 'ctg_part_cpu', 2);

INSERT INTO `product`
            (`p_category`, `p_name`, `p_price`)
VALUES      (1, 'Дешёвый принтер', 100),
            (1, 'Дорогой принтер', 1000),
            (1, 'Просто принтер', 300),
            (2, 'Дешёвый процессор', 500),
            (2, 'Дорогой процессор', 7000);

INSERT INTO `ctg_part_printer`
            (`cp_id`, `cp_type`, `cp_ppm`)
VALUES      (1, 'laser', 10),
            (2, 'laser', 30),
            (3, 'ink-jet', 5);

INSERT INTO `ctg_part_cpu`
            (`cp_id`, `cp_frequency`, `cp_core_count`)
VALUES      (4, 1800, 2),
            (5, 3500, 6);
```

На что здесь стоит обратить внимание:

- первичные ключи в таблицах **`ctg_part_printer`** и **`ctg_part_cpu`** не автоинкрементируемые (их значение заранее известно, т.к. они являются как первичными, так и внешними ключами);
- при вставке данных о товаре обязательно соблюдать последовательность
  1. вставка записи в таблицу **`product`**;
  2. выяснение полученного значения её автоинкрементируемого первичного ключа;
  3. вставка записи в дочернюю таблицу (с дополнительным описанием товара) с использованием выясненного значения первичного ключа.
- такую последовательность действий обязательно нужно выполнять в рамках одной сессии (не закрывая соединение с СУБД), т.к. в противном случае можно «потерять» новое значение первичного ключа таблицы **`product`** и/или «прикрепить» дополнительное описание товара к неправильному товару;
- в таблице **`category`** есть специальное поле **`c_table`**, хранящее имя таблицы, в которой расположены дополнительные данные о товарах каждой категории (имя этой таблицы обязательно нужно знать для выполнения подавляющего большинства вопросов в базе данных, построенной по такой модели).

Итоговая картина того, как выглядит база данных, построенная по представленной на рисунке 2.3.w схеме, созданная и наполненная данными с помощью только что рассмотренного кода, показана на рисунке 2.3.x.

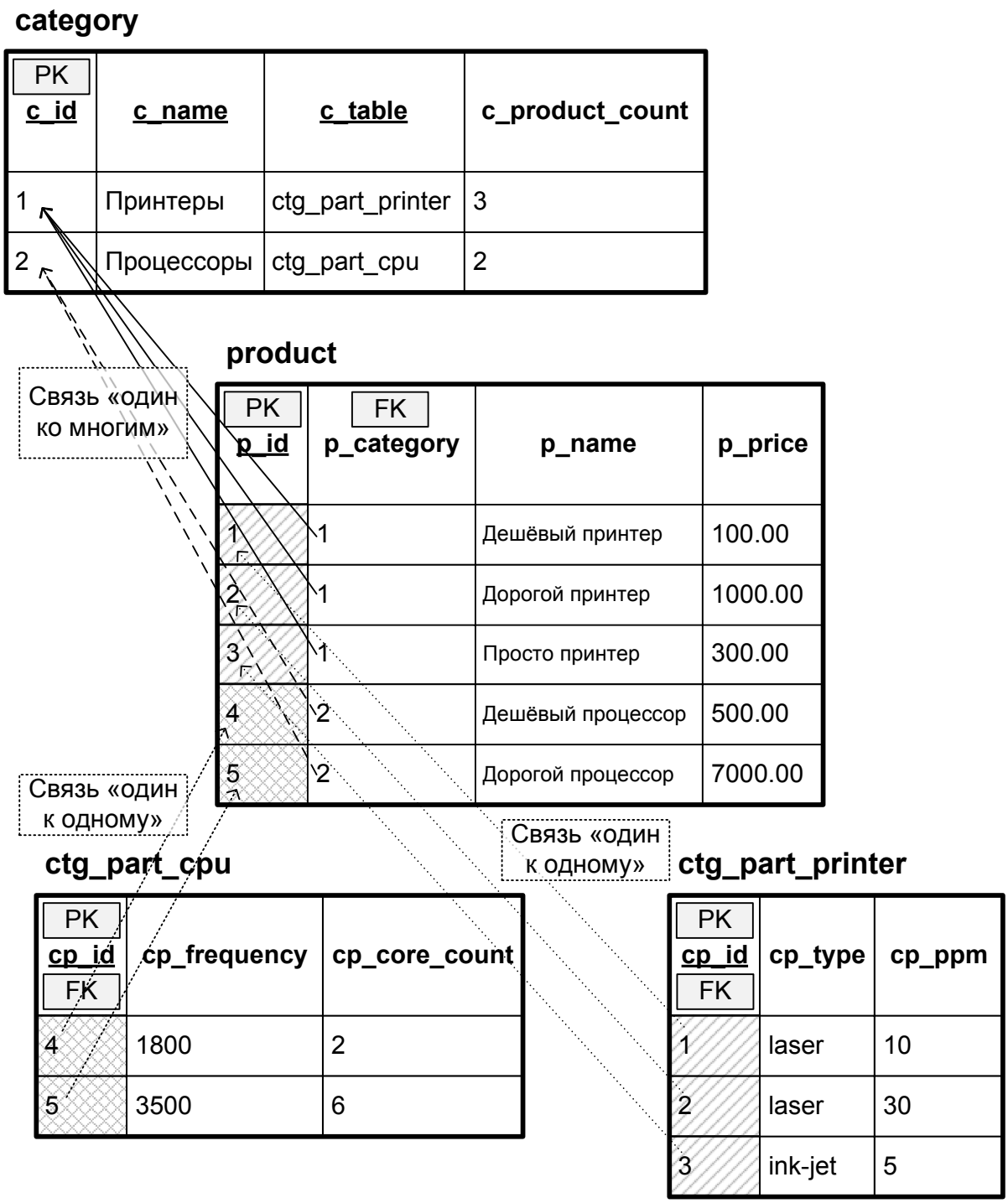


*Рисунок 2.3.x — **Итоговое представление фрагмента базы данных, построенного с использованием связи «один к одному»***

Перед тем, как приступить к рассмотрению операций с такой базой данных, ещё раз подчеркнём суть: описание каждого товара разбито на две части, одна из которых (характерная для любого товара) хранится в таблице `product`, а вторая (характерная только для товаров данной категории) хранится в своей отдельной таблице.

Выполнение вставки данных может быть реализовано двумя способами:

- отдельным запросом выполнить вставку записи в таблицу **product**, отдельным запросом выяснить новое значение автоинкрементируемого первичного ключа, отдельным запросом выполнить вставку записи в дочернюю таблицу (подставив туда полученное на предыдущем шаге значение первичного ключа) — каждый из таких запросов, как правило, по-отдельности выполняется на уровне приложения, работающего с базой данных;
- создать хранимую процедуру{363}, которая будет получать набор данных для вставки и все необходимые операции выполнять «внутри себя» — это более надёжный способ, но такую хранимую процедуру придётся создавать отдельно для каждой категории товаров (или писать довольно сложное решение с передачей в процедуру и последующим анализом массива данных).

MySQL **Пример кода к рисунку 2.3.x (вставка данных, вариант 1 — последовательное выполнение операций)**

```
-- а) Вставка данных в таблицу "product":
INSERT INTO `product`
            (`p_category`, `p_name`, `p_price`)
VALUES      (1, 'Новый принтер', 150.43);

-- б) Определение нового значения автоинкрементируемого первичного ключа:
SET @new_key = (SELECT LAST_INSERT_ID());

-- в) Вставка записи в таблицу, хранящую специфические свойства товара:
INSERT INTO `ctg_part_printer`
            (`cp_id`, `cp_type`, `cp_ppm`)
VALUES      (@new_key, 'laser', 17);
```

MySQL **Пример кода к рисунку 2.3.x (вставка данных, вариант 2 — создание хранимой процедуры)**

```
DROP PROCEDURE ADD_PRINTER;
DELIMITER $$
CREATE PROCEDURE
 ADD_PRINTER (IN printer_name  VARCHAR(50),
              IN printer_price DECIMAL(10, 2),
              IN printer_type  VARCHAR(50),
              IN printer_ppm   SMALLINT,
              OUT new_pk INT)
BEGIN
 -- Определение идентификатора категории. В реальности этого можно
 -- не делать (маловероятно, что он поменяется), но здесь эта
 -- операция приведена для демонстрации полного набора действий.
 SET @category_id = (SELECT `c_id`
                     FROM   `category`
                     WHERE  `c_table` = 'ctg_part_printer');

 -- Вставка данных в таблицу "product":
 INSERT INTO `product`
             (`p_category`, `p_name`, `p_price`)
 VALUES      (@category_id, printer_name, printer_price);

 -- Определение нового значения автоинкрементируемого первичного ключа:
 SET @new_key = (SELECT LAST_INSERT_ID());

 -- Вставка записи в таблицу, хранящую специфические свойства товара:
 INSERT INTO `ctg_part_printer`
             (`cp_id`, `cp_type`, `cp_ppm`)
 VALUES      (@new_key, printer_type, printer_ppm);

 SET new_pk = @new_key;
END;
$$
DELIMITER ;
```

MySQL **Пример кода к рисунку 2.3.x (вставка данных, вариант 2 — использование хранимой процедуры)**

```
CALL ADD_PRINTER('Совсем новый принтер', 199.23, 'photo', 2, @pk_assigned);
SELECT @pk_assigned;
```

После выполнения хранимой процедуры мы как «приятный бонус» получаем значение первичного ключа записи, хранящей данные о товаре (это значение одинаково для таблиц **product** и **ctg_part_printer**, потому нет необходимости уточнять, о какой из таблиц идёт речь).

Переменную, с помощью которой мы получаем это значение, также можно использовать для передачи информации о возникших ошибках (например, используя диапазон отрицательных целых чисел как коды ошибок). В настоящий момент это не было реализовано в рассмотренном коде хранимой процедуры с целью его максимального упрощения.

В отличие от вставки данных, которая требует столь необычного подхода, обновление и удаление происходит совсем тривиально. При обновлении просто стоит помнить, что данные о товаре представлены в двух различных таблицах (т.е. надо обновить нужную из них или обе). А для выполнения удаления нужно удалить запись из таблицы product (что приведёт к активации каскадной операции и автоматическому удалению соответствующей записи из соответствующей подчинённой таблицы).

Приведём соответствующие примеры кода:

MySQL **Пример кода к рисунку 2.3.x (обновление и удаление данных)**

```
-- Обновление части данных, расположенных в таблице "product":
UPDATE `product`
   SET `p_price` = 299.45
 WHERE p_id = 7;

-- Обновление части данных, расположенных в таблице "ctg_part_printer":
UPDATE `ctg_part_printer`
   SET `cp_ppm` = 3
 WHERE cp_id = 7;

-- Удаление данных:
DELETE FROM `product`
      WHERE p_id = 7;
```

Все операции модификации данных мы рассмотрели. Осталось продемонстрировать, как при использовании такой схемы происходит выборка данных.

Здесь задача разделяется на две:

- в простом случае запрос будет выполняться к таблице **product** и/или таблицам **product** и **category** (например, для подсчёта средней цены всех товаров или средней цены товаров в рамках отдельной категории товаров) — здесь логика ничем не отличается от рассмотренных ранее примеров работы со связями «один ко многим»;
- в более сложном случае запрос нужно будет выполнить к таблице **product** и соответствующей таблице (зависящей от категории товара), в которой хранятся дополнительные специфические параметры товаров.

Для первого случая просто рассмотрим пару примеров запросов без пояснений (см. ещё раз выше{78} в данной главе описание работы со связями типа «один ко многим», если что-то в этих запросах вызывает сложность).

Представленные ниже запросы позволяют вычислить среднюю стоимость всех товаров (запрос только к таблице **product**) и построить список категорий товаров с указанием средней цены товаров в каждой из них (запрос к таблицам **product** и **category**).

MySQL Пример кода для определения средней стоимости всех товаров и товаров по каждой категории

```
-- Определение средней цены товаров:
SELECT AVG(`p_price`) AS `avg_price`
FROM   `product`

-- Определение средней цены товаров по каждой категории:
SELECT `c_name`,
       AVG(`p_price`) AS `avg_price`
FROM   `product`
       JOIN `category`
         ON `p_category` = `c_id`
GROUP  BY `p_category`
```

Результат выполнения первого запроса:

avg_price
1780.000000

Результат выполнения второго запроса:

c_name	avg_price
Принтеры	466.666667
Процессоры	3750.000000

Если же мы хотим получить полный список (со всеми параметрами) товаров из некоторой категории, придётся выполнить запрос к таблице **product** и соответствующей дочерней таблице, хранящей данные о товарах указанной категории.

Отметим, что технологически сложно (и практически бессмысленно) пытаться получить полный список (со всеми параметрами) всех товаров нескольких (или всех сразу) категорий одновременно, т.к. в результирующей таблице данные в столбцах, описывающих специфические параметры, будут иметь разный смысл для разных категорий товаров.

Это очень типичная ошибка, приводящая к нарушению фундаментальных основ реляционной теории: по определению данные в одном столбце таблицы (атрибуте отношения) должны иметь один и тот же смысл, т.е. одинаково трактоваться для всех записей таблицы (кортежей отношения). В рассматриваемом же случае это правило нарушается, и выглядеть это будет так.

MySQL Пример кода к рисунку 2.3.x (обновление и удаление данных)

```
SELECT `p_id`,
       `p_name`,
       `p_price`,
       `cp_frequency`  AS `param1`,
       `cp_core_count` AS `param2`
FROM   `product`
       JOIN `ctg_part_cpu`
         ON `p_id` = `cp_id`
UNION
SELECT `p_id`,
       `p_name`,
       `p_price`,
       `cp_type` AS `param1`,
       `cp_ppm`  AS `param2`
FROM   `product`
       JOIN `ctg_part_printer`
         ON `p_id` = `cp_id`
```

В результате выполнения такого запроса получится результат, в котором значения полей **`param1`** и **`param2`** носят разный смысл для разных строк (так, например, поле **`param1`** для принтеров означает тип, а для процессоров — частоту).

p_id	p_name	p_price	param1	param2
4	Дешёвый процессор	500.00	1800	2
5	Дорогой процессор	7000.00	3500	6
1	Дешёвый принтер	100.00	laser	10
2	Дорогой принтер	1000.00	laser	30
3	Просто принтер	300.00	ink-jet	5

Иногда можно услышать контраргумент, состоящий в том, что почти любая СУБД позволяет выполнять запросы с группировками и подведением итогов (**`ROLLUP`**, **`CUBE`**), где значения полей в некоторых строках принимают иной особый смысл. Да, СУБД умеют технически выполнять много операций, потенциально приводящих к множеству ошибок в приложениях. Пользоваться ли этими возможностями, или отдать предпочтение надёжности — решать вам.

Возвращаемся к поставленной задаче. Как и в других рассмотренных выше примерах, она имеет несколько решений.

Первое (простое) осуществимо тогда, когда на стороне работающего с СУБД приложения мы знаем имя таблицы, хранящей специфические параметры указанной категории товаров. Тогда решение можно выполнить одним простым запросом:

MySQL **Пример кода случая, когда запрос полностью формируется на стороне приложения**

```
SELECT `product`.*,
       `ctg_part_cpu`.*
FROM   `product`
       JOIN `ctg_part_cpu`
         ON `p_id` = `cp_id`
```

В результате выполнения этого запроса получается следующий результат (да, запрос можно улучшить, перечислив конкретные интересующие нас поля таблиц, но в данном случае в целях наглядности выбран полный набор данных):

p_id	p_category	p_name	p_price	cp_id	cp_frequency	cp_core_count
4	2	Дешёвый процессор	500.00	4	1800	2
5	2	Дорогой процессор	7000.00	5	3500	6

Однако, хотелось бы получить возможность произвольно выбирать данные по некоторой категории товаров, просто указав её идентификатор и более ни о чём не думая. Поскольку MySQL не позволяет выполнять внутри хранимых функций{363} динамические запросы, решение построим на основе хранимой процедуры{363}.

В рассмотренном ниже коде все те действия, которые должно было бы выполнить работающее с базой данных приложение, реализованы внутри хранимой процедуры:

- сначала на основе идентификатора категории товаров мы получаем имя таблицы, в которой хранятся специфические параметры товаров указанной категории;
- далее у нас есть два варианта развития событий:
  - такой категории товаров нет — нужно вернуть пустой результат;
  - такая категория товаров есть, мы знаем имя таблицы, в которой хранятся специфические параметры её товаров, и можем сформировать и выполнить запрос для получения этой информации.

MySQL Создание хранимой процедуры для получения всех данных о товарах заданной категории

```
DELIMITER $$
CREATE PROCEDURE
 SHOW_PRODUCTS_FROM_CATEGORY (IN category_id INT)
BEGIN
 -- Определение имени таблицы, в которой хранятся данные
 -- о специфических параметрах товаров данной категории:
 SET @ctg_table = (SELECT `c_table`
                   FROM   `category`
                   WHERE  `c_id` = category_id);

 IF (@ctg_table IS NULL)
  THEN
   -- Если категория с искомым идентификатором не найдена,
   -- происходит возврат пустого значения.
   SELECT NULL;
  ELSE
   -- Формирование текста запроса:
   SET @query_text = CONCAT('SELECT `product`.*, `', @ctg_table,
                            '`.* FROM `product` JOIN `', @ctg_table,
                            '` ON `p_id` = `cp_id`');

   -- Подготовка и выполнение запроса:
   PREPARE query_stmt FROM @query_text;
   EXECUTE query_stmt;
  END IF;
END;
$$
DELIMITER ;
```

Теперь получение полного набора данных о товарах любой категории сводится к вот такому простому вызову хранимой процедуры:

MySQL Использование хранимой процедуры для получения всех данных о товарах заданной категории

```
CALL SHOW_PRODUCTS_FROM_CATEGORY(2);
```

Последнее, что нам осталось рассмотреть в данной главе — реализацию обеспечения консистентности данных с помощью триггеров (о которых будет подробно рассказано в соответствующем разделе{350}, потому здесь будет лишь пример кода с минимальными пояснениями).

Этот пример не относится ни к какому из видов связей в отдельности и может применяться с любыми связями, кроме «один к одному», где он теряет смысл.

Возможно, вы заметили, что в таблице `category` есть поле `c_product_count`, значение которого для каждой записи равно количеству товаров в соответствующей категории. Конечно, его можно в любой момент вычислить отдельным запросом, но для случая очень большого количества товаров и их категорий выполнение такого запроса может занять ощутимое время, и потому хотелось бы иметь возможность получить интересующее нас значение мгновенно — именно потому мы и храним его явно.

При наполнении таблиц тестовыми данными мы указывали это значение в самом запросе на вставку, т.к. мы его знали заранее, но в реальной работе базы данных оно должно вычисляться автоматически, что представляет собой классический вариант обеспечения консистентности{72} базы данных.

Значение этого поля должно меняться в трёх случаях:

- в категорию добавлен новый товар (вставка данных);
- из категории удалён товар (удаление данных);
- товар перемещён из одной категории в другую (обновление данных).

Иными словами, СУБД должна автоматически обновлять значение этого поля во всех трёх перечисленных случаях. И у нас есть соответствующий механизм, позволяющий полностью автоматизировать данную операцию — триггеры{350}.

Как вы уже могли догадаться, и эта задача также имеет несколько решений (а если учесть особенности разных СУБД, вариантов будет ещё больше). В общем можно выделить два подхода:

- надёжный, но медленный — можно заново подсчитывать количество товаров в каждой категории при выполнении модификации списка товаров;
- быстрый, но менее надёжный — можно изменять значение количества товаров в каждой категории на то количество товаров, которое было добавлено/удалено/перемещено (здесь мы рискуем получить некорректное значение, если исходное значение не соответствовало действительности).

Реализуем оба варианта и начнём с медленного (и надёжного). Основной код триггеров на вставку и удаление можно было также реализовать через запрос на обновление с `JOIN`, но для разнообразия там реализовано альтернативное решение.

MySQL **Создание триггеров (первый вариант — более медленный, но более надёжный)**

```
DELIMITER $$
CREATE TRIGGER `prod_count_ins`
AFTER INSERT ON `product`
 FOR EACH ROW
  BEGIN
   UPDATE `category`
   SET    `c_product_count` = (SELECT COUNT(`p_id`)
                               FROM   `product`
                               WHERE  `p_category` = NEW.`p_category`)
   WHERE  `c_id` = NEW.`p_category`;
  END;
$$

CREATE TRIGGER `prod_count_upd`
AFTER UPDATE ON `product`
 FOR EACH ROW
  BEGIN
   UPDATE `category`
    JOIN (SELECT `p_category`,
                 COUNT(`p_id`) AS `new_count`
          FROM   `product`
          GROUP  BY `p_category`
          HAVING `p_category` IN (OLD.`p_category`, NEW.`p_category`)
         ) AS `prepared_data`
         ON `c_id` = `p_category`
   SET  `c_product_count` = `new_count`
   WHERE `c_id` IN (OLD.`p_category`, NEW.`p_category`);
  END;
$$

CREATE TRIGGER `prod_count_del`
AFTER DELETE ON `product`
 FOR EACH ROW
  BEGIN
   UPDATE `category`
   SET    `c_product_count` = (SELECT COUNT(`p_id`)
                               FROM   `product`
                               WHERE  `p_category` = OLD.`p_category`)
   WHERE  `c_id` = OLD.`p_category`;
  END;
$$
DELIMITER ;
```

Подробное пояснение логики работы подобных триггеров приведено в соответствующей книге[64], но если изложить суть вкратце, получается:

- в первом триггере на основе значения идентификатора категории добавленного товара `NEW.`p_category`` в подзапросе определяется новое количество товаров в данной категории, и это значение используется для обновления поля `c_product_count`;
- во втором триггере необходимо обновить значение поля `c_product_count` для двух категорий (идентификаторы которых находятся в `OLD.`p_category`` и `NEW.`p_category``) — чтобы не выполнять два отдельных запроса, здесь мы использовали запрос на обновление с `JOIN`;
- в третьем триггере на основе значения идентификатора категории удалённого товара `OLD.`p_category`` в подзапросе определяется оставшееся количество товаров в данной категории, и это значение используется для обновления поля `c_product_count`.

Обратите внимание на тот факт, что все триггеры — именно `AFTER`-триггеры, т.е. они должны срабатывать **после** выполнения операции модификации, чтобы обрабатывать новые актуальные данные.

Переходим ко второму варианту решения — быстрому, но не гарантирующему корректность полученного результата (в случае, если исходное значение поля `c_product_count` было неверным).

MySQL **Создание триггеров (второй вариант — более быстрый, но менее надёжный)**

```
DELIMITER $$
CREATE TRIGGER `prod_count_ins_fast`
AFTER INSERT ON `product`
 FOR EACH ROW
  BEGIN
   UPDATE `category`
   SET    `c_product_count` = `c_product_count` + 1
   WHERE  `c_id` = NEW.`p_category`;
  END;
$$

CREATE TRIGGER `prod_count_upd_fast`
AFTER UPDATE ON `product`
 FOR EACH ROW
  BEGIN
   UPDATE `category`
   SET    `c_product_count` = `c_product_count` + 1
   WHERE  `c_id` = NEW.`p_category`;
   UPDATE `category`
   SET    `c_product_count` = `c_product_count` - 1
   WHERE  `c_id` = OLD.`p_category`;
  END;
$$

CREATE TRIGGER `prod_count_del_fast`
AFTER DELETE ON `product`
 FOR EACH ROW
  BEGIN
   UPDATE `category`
   SET    `c_product_count` = `c_product_count` - 1
   WHERE  `c_id` = OLD.`p_category`;
  END;
$$
DELIMITER ;
```

[64] «Работа с MySQL, MS SQL Server и Oracle в примерах» (С.С. Куликов), пример 32 [http://svyatoslav.biz/database_book/]

Код этих триггеров получается намного проще, чем в первом варианте решения. Вся суть операции сводится к тому, чтобы узнать идентификаторы `NEW.`p_category`` (в такую категорию товар был добавлен) и `OLD.`p_category`` (из такой категории товар был удалён), после чего значение поля `c_product_count` соответствующей категории увеличивается или уменьшается на единицу (в зависимости от того, был ли товар добавлен или удалён).

Остаётся проверить, как работают рассмотренные триггеры (очевидно, нужно создавать набор триггеров **или** по первому варианту, **или** по второму варианту, но **не** одновременно).

MySQL Проверка работоспособности `INSERT`-триггера

```
INSERT INTO `product`
            (`p_category`, `p_name`, `p_price`)
VALUES      (1, 'Какой-то принтер 1', 200.5),
            (1, 'Какой-то принтер 2', 300.7)
```

			Было	Стало
c_id	**c_name**	**c_table**	**c_product_count**	**c_product_count**
1	Принтеры	ctg_part_printer	3	5
2	Процессоры	tg_part_cpu	2	2

MySQL Проверка работоспособности `UPDATE`-триггера

```
UPDATE `product`
SET    `p_category` = 2
WHERE  `p_id` = 1
```

			Было	Стало
c_id	**c_name**	**c_table**	**c_product_count**	**c_product_count**
1	Принтеры	ctg_part_printer	5	4
2	Процессоры	ctg_part_cpu	2	3

MySQL Проверка работоспособности `DELETE`-триггера

```
DELETE FROM `product`
WHERE  `p_id` > 5
```

			Было	Стало
c_id	**c_name**	**c_table**	**c_product_count**	**c_product_count**
1	Принтеры	ctg_part_printer	4	2
2	Процессоры	ctg_part_cpu	3	3

На этом основные операции, связанные с практическим использованием связей, можно считать рассмотренными. Хотя, конечно, в следующих разделах мы ещё неоднократно к ним вернёмся, т.к. сложно представить более фундаментальную для работы с реляционными базами данных тему.

Задание 2.3.g: реализуйте на практике идею, описанную выше в примечании{90} о случаях, когда в «таблице связи» необходимо учитывать количество установленных связей. Для управления счётчиком используйте триггеры. Примеры предметных областей, для которых актуальна такая схема (но лучше — придумайте свою):

- пользователи и музыкальные композиции (счётчик хранит количество прослушиваний);
- владельцы дисконтных карт и магазины (счётчик хранит количество посещений);
- студенты и предметы (счётчик хранит количество посещений).

Задание 2.3.h: представьте, что в некотором приложении нужно отслеживать и фиксировать, сколько сообщений пользователи отправили друг другу (любой пользователь может отправить любому другому пользователю (но не самому себе) любое количество сообщений). Создайте фрагмент схемы базы данных для хранения соответствующей информации и реализуйте ограничение, не позволяющее внести в базу данных факт отправки пользователем сообщения самому себе.

Задание 2.3.i: стоит ли в базе данных «Банк»{408} в каких бы то ни было таблицах, реализующих связи «многие ко многим», реализовать описанное ранее решение{90} со «счётчиком связей»? Если вы считаете, что «да», внесите соответствующие правки в модель.

2.4. ИНДЕКСЫ

2.4.1. ОБЩИЕ СВЕДЕНИЯ ОБ ИНДЕКСАХ

В полном развёрнутом виде материал этой главы должен занимать несколько тысяч страниц. Потому просьба принять во внимание, что здесь будет представлена лишь самая основная информация, необходимая для формирования представления о том, как работают индексы.

Также стоит отметить, что индексы почти полностью лежат в области физического уровня{314} проектирования базы данных, и реляционная теория их почти не затрагивает.

В общем виде определение понятия «индекс» может быть сформулировано следующим образом:

Индекс (index[65]) — специальная структура базы данных, используемая для ускорения поиска записей и физического доступа к записям.

Упрощённо: механизм, значительно ускоряющий поиск необходимой информации в базе данных (по аналогии: для человека карта города является таким «индексом», позволяющим быстро найти нужное здание).

Индексы широко используются при проектировании баз данных в силу следующих преимуществ:

- размер индексов значительно меньше размера индексированных данных, что позволяет размещать их в оперативной памяти для ускорения доступа;
- структура индексов специальным образом оптимизирована для выполнения операций поиска;
- как следствие двух предыдущих пунктов, индексы значительно (иногда — на несколько порядков) ускоряют поиск данных.

Однако, у индексов есть и недостатки, которые обязательно следует учитывать, чтобы не создавать индексы там, где они не нужны:

- когда индексов становится много, они занимают ощутимый объём оперативной памяти;
- наличие индексов ощутимо замедляет операции модификации данных (вставки, удаления, обновления), т.к. при изменении данных СУБД необходимо обновить индексы, приведя их в соответствие с новыми значениями индексированных данных.

Отсюда логически вытекает вопрос: как понять, что тот или иной индекс нужен? Есть несколько признаков, говорящих в пользу создания индекса:

- операции чтения из таблицы выполняются гораздо чаще, чем операции модификации;
- поле или совокупность полей часто фигурируют в запросах в секции `WHERE`;
- исследование показало, что наличие индекса повышает производительность запросов;
- необходимо обеспечить уникальность значения поля (или совокупности полей), не являющегося первичным ключом (в таком случае строится т.н. уникальный индекс{109});
- поле (или совокупность полей) является внешним ключом — в таком случае индексы могут значительно ускорить выполнение `JOIN`-запросов.

65 **Index** — a specific kind of physical access path (an implementation construct, intended to improve the speed of access to data as physically stored). («The New Relational Database Dictionary», C.J. Date)

Разновидностей индексов очень много, и сейчас мы рассмотрим основные примеры (см. рисунок 2.4.а). Также отметим, что в зависимости от СУБД и выбранного метода доступа{33} список поддерживаемых индексов и особенности их реализации могут очень сильно различаться.

*Рисунок 2.4.а — **Виды индексов и их взаимосвязь***

По количеству полей индексы классифицируются совершенно аналогично ключам{35}, а именно:

Простой индекс (simple index, single-column index[66]) — индекс, построенный на одном поле таблицы.

Упрощённо: индекс, включающий информацию о содержимом только одного поля таблицы.

Составной индекс (composite index[67]) — индекс, построенный на двух и более полях таблицы.

Упрощённо: индекс, включающий информацию о содержимом двух и более полей таблицы.

В некоторых случаях (с определённой долей допущения) понятия «простой ключ{40}»/«простой индекс» и «составной ключ{40}»/«составной индекс» могут выступать как синонимы[68].

Как простой, так и составной индексы могут использоваться для обеспечения уникальности значений альтернативных ключей{38} (тогда это будет уникальный индекс{109}) или ускорения поиска записей по значению индексированных полей (такой индекс может быть как уникальным{109}, так и неуникальным{109}).

Для составных индексов в полной мере характерна подробно рассмотренная ранее{52} проблема последовательности полей: то поле, по которому поиск часто будет выполняться отдельно от других полей, должно быть первым.

Любая современная СУБД поддерживает использование как простых, так и составных индексов.

Поскольку очень часто звучит вопрос о том, как составной индекс организован физически, поясним этот момент на примере сравнения простого и составного индекса на основе сбалансированного дерева{117}. На рисунке 2.4.b структура такого дерева представлена упрощённо, а более подробное пояснение приведено на рисунке 2.4.i далее{119}.

В общем случае для индексирования второго (третьего и т.д.) полей применяется две стратегии:

- построение комбинированных ключей, когда значения дополнительных полей хранятся рядом со значениями основных полей — такой подход технически проще, но может привести к ощутимому увеличению затрат памяти, если в хранимых данных наблюдается ситуация, при которой одному значению первого индексируемого поля соответствует много различных значений последующих полей;
- использование вложенных деревьев (фактически, каждый узел основного дерева является корнем дополнительного вложенного дерева) — такой подход технически сложнее, но позволяет сэкономить память в случае, когда одному значению первого индексируемого поля соответствует много различных значений последующих полей.

---

66 **Single-column index** — an index that is created based on only one table column («SQL Indexes»). [https://www.tutorialspoint.com/sql/sql-indexes.htm]

67 **Composite index** — an index on two or more columns of a table («SQL Indexes»). [https://www.tutorialspoint.com/sql/sql-indexes.htm]

68 Говоря чуть строже, «ключ» подразумевает уникальность значений в столбце, а «индекс» может быть и «неуникальным», т.е. не требовать соблюдения этого правила.

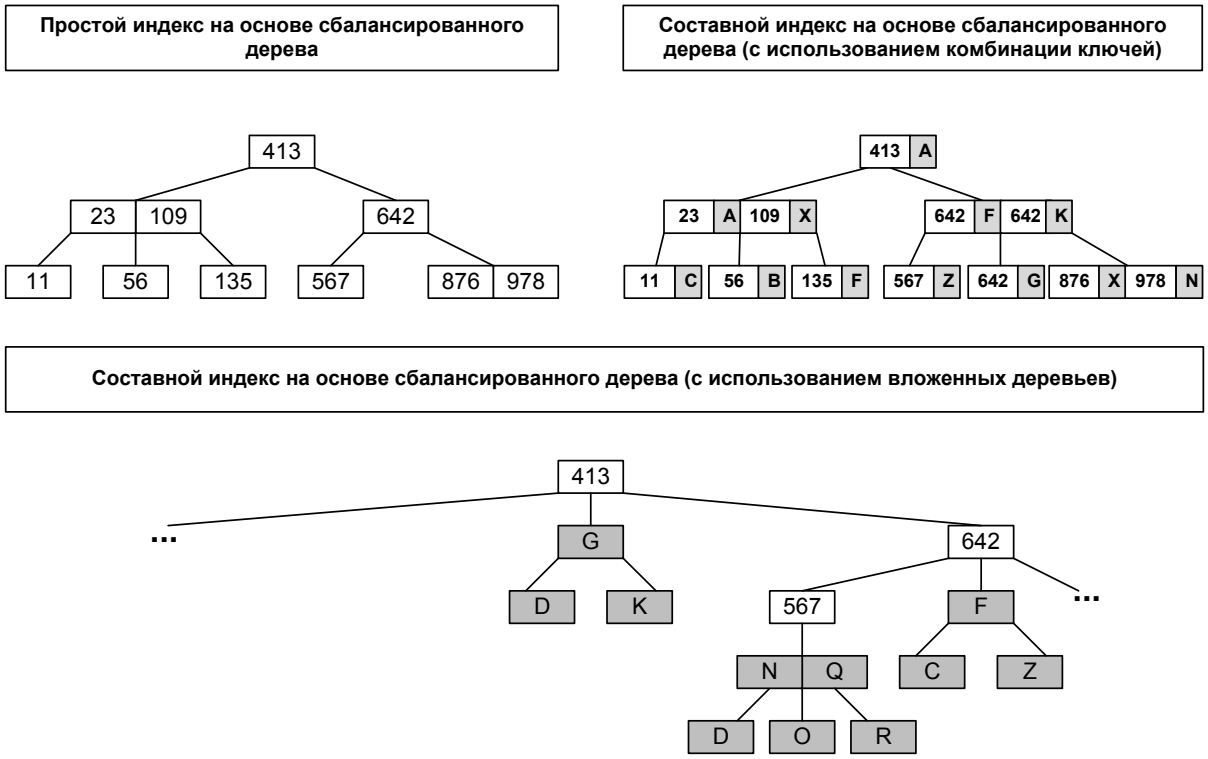


*Рисунок 2.4.b — **Структура простого и составного индекса на основе B-дерева***

По уникальности записей индексы также можно сравнить с ключами{35} в том смысле, что понятия «уникальный индекс» и «ключ» даже на уровне синтаксиса SQL некоторыми СУБД рассматриваются как синонимы. В то же время стоит помнить, что реляционная теория оперирует понятием «ключ», а «индекс» появляется на уровне реализации базы данных в конкретной СУБД.

Уникальный индекс (unique index[69]) — индекс, построенный на содержащем уникальные значения поле (полях) таблицы.

Упрощённо: индекс, построенный на поле, являющемся первичным или альтернативным ключом таблицы.

Неуникальный индекс (non-unique index, index[70]) — индекс, построенный на **не** содержащем уникальные значения поле (полях) таблицы.

Упрощённо: «просто индекс» (для неуникального индекса в синтаксисе SQL даже нет отдельного ключевого слова, просто пишется ***`INDEX`****).*

Как уникальный, так и неуникальный индекс может быть простым{108} или составным{108}.

Задачей неуникального индекса является только ускорение операций поиска по индексируемым полям, в то время как уникальный индекс связан с контролем уникальности значений индексируемого поля (полей).

Разные СУБД в этом контексте предлагают разные возможности:

- в MySQL единственным способом обеспечения гарантированной уникальности значений поля (полей), не являющегося первичным ключом, является создание на этом поле (полях) уникального индекса;

---

69 **Unique index** — an index on the basis of some superkey.

70 **Non-unique index, index** — см. index{106}.

- в MS SQL Server включение свойства уникальности значения поля приводит к созданию на этом поле уникального индекса (и наоборот — создание на поле уникального индекса приводит к включению свойства уникальности значений этого поля);
- в Oracle уникальность значений поля может быть обеспечена как с помощью уникального индекса, так и без него, но из соображений повышения производительности рекомендуется строить на поле с уникальными значениями уникальный индекс.

Любая современная СУБД поддерживает использование как уникальных, так и неуникальных индексов.

По соотношению с расположением записей индексируемой таблицы индексы делятся на кластерные (и первичные — как их подвид) и некластерные.

Кластерный индекс (clustered index[71]) — индекс, построенный на поле (возможно, с неуникальными значениями), по которому произведено физическое упорядочивание данных в файле.

Упрощённо: по значению индексированного поля данные таблицы физически отсортированы на диске; значения поля могут повторяться.

Первичный индекс (primary index[72]) — индекс, построенный на поле с уникальными значениями, по которому произведено физическое упорядочивание данных в файле.

*Упрощённо: по значению индексированного поля данные таблицы физически отсортированы на диске; значения поля **не** могут повторяться.*

Некластерный индекс (non-clustered index[73]) — индекс, построенный на поле, по которому **не** произведено физическое упорядочивание данных в файле.

Упрощённо: «просто индекс», принципы упорядочивания в котором никак не связаны с физическим расположением данных на диске.

Для начала поясним, как связаны между собой «первичный ключ{39}» и «первичный индекс{110}».

В теории «первичный ключ» относится к связям{57}, ссылочной целостности{72} и нормализации{161} — т.е. теоретическим основам реляционных баз данных, а «первичный индекс» относится к способу организации данных на диске и методам доступа{33} — т.е. к физическому уровню проектирования и способам реализации конкретной СУБД.

На практике в различных СУБД ситуация такова:

- MySQL всегда строит первичный индекс на первичном ключе;
- MS SQL Server позволяют построить «кластерный уникальный» (фактически — первичный) индекс на поле (полях), не являющемся первичным ключом;
- Oracle позволяют использовать для индексации первичного ключа даже неуникальные индексы, а упорядочивание данных таблицы (именно в такой формулировке, т.е. Oracle не оперирует понятием «кластерного индекса», а явно говорит об «индексно-организованной таблице») по первичному ключу не производить.

---

71 **Clustered index** — an index that is used when ordering field is not a key field — that is, if numerous records in the file can have the same value for the ordering field. (“Fundamentals of Database Systems”, Ramez Elmasri, Shamkant Navathe)

72 **Primary index** — an index that is specified on the ordering key field of an ordered file of records. (“Fundamentals of Database Systems”, Ramez Elmasri, Shamkant Navathe)

73 **Non-clustered index** — см. index{106}.

Вкратце: чаще всего «первичный ключ» и «первичный индекс» будут синонимами, но исключения возможны (и нередки).

Теперь посмотрим, как это выглядит. Начнём с кластерного и первичного индексов (рисунок 2.4.c).

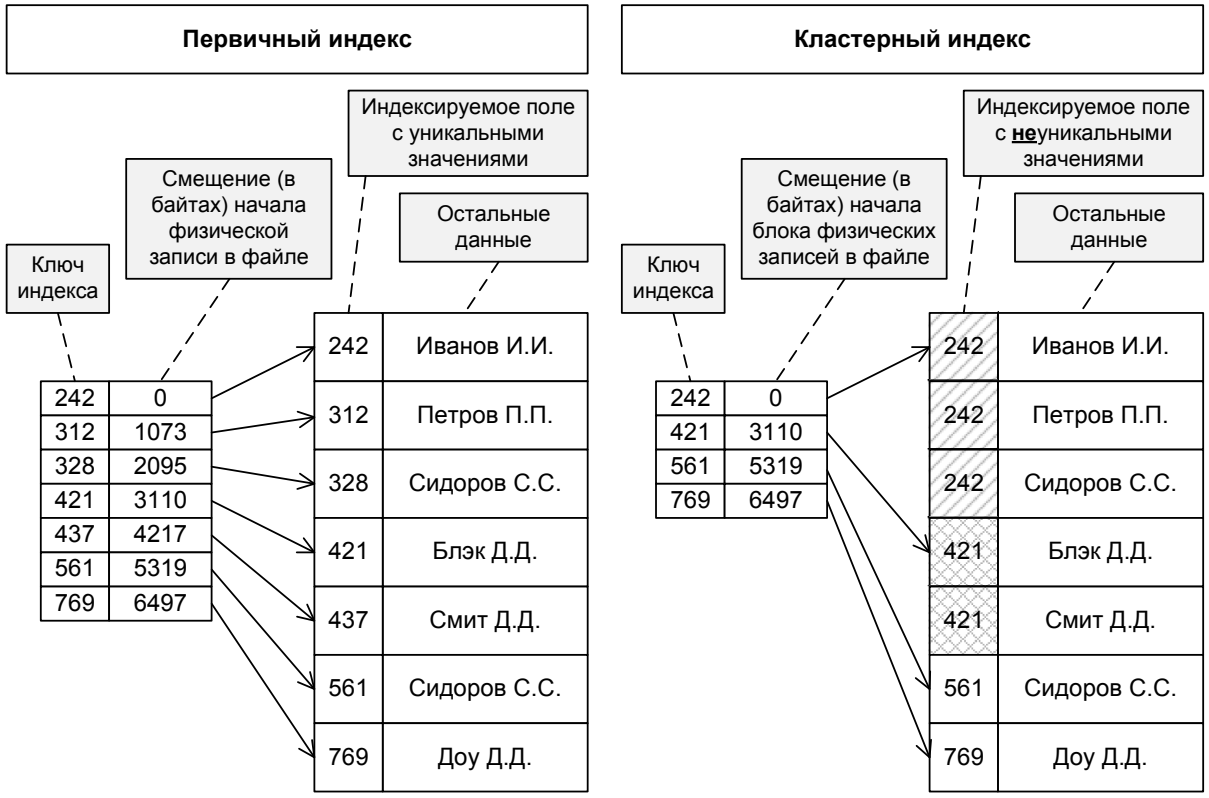


*Рисунок 2.4.c — **Схематичное представление первичного и кластерного индекса***

Ключевое отличие состоит в том, что первичный индекс содержит данные о расположении каждой записи с **уникальным значением индексируемого поля**, а кластерный — о начале блока записей с **одинаковыми значениями индексируемого поля**. Если не вдаваться в технические особенности реализации алгоритмов построения и использования таких индексов — в остальном они идентичны.

Первичный индекс также может содержать данные о расположении не каждой записи, а блока записей, см. далее «неплотный индекс{115}».

Некластерные же индексы отличаются от кластерных тем, что последовательность записей в индексе и физическом файле не совпадает. Как правило, это приводит к усложнению структуры самого индекса из-за необходимости хранения нескольких разных адресов записей с совпадающими значениями индексируемого поля.

Схематичное представление некластерного индекса представлено на рисунке 2.4.d.

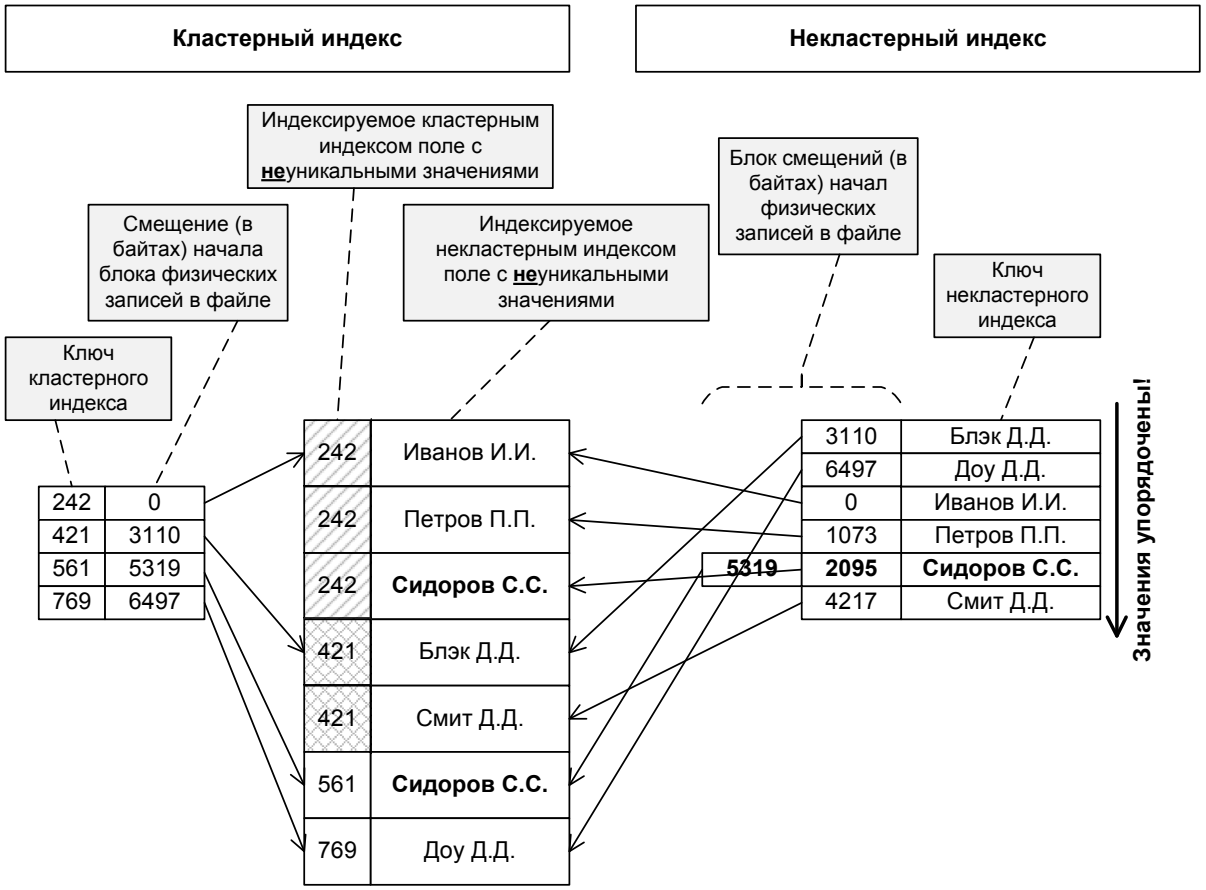


*Рисунок 2.4.d — **Схематичное представление некластерного индекса (в сравнении с кластерным)***

Как видно на рисунке 2.4.d, ключи некластерного индекса упорядочены (в данном случае — по алфавиту) по значению индексируемого поля, но сами данные в таблице остались упорядоченными по другому полю (на котором построен кластерный индекс).

В случае неуникальности значений индексируемого поля (у нас таким значением является «Сидоров С.С.» в некластерном индексе необходимо строить блок (цепочку) смещений для каждой записи в таблице, индексируемое поле которой содержит соответствующее значение.

Любая современная СУБД поддерживает использование как кластерных, так и некластерных индексов, но в некоторых случаях (например, в MySQL) кластерный индекс строится автоматически (на первичном ключе, или первом уникальном индексе, или специально добавленном поле[74]), т.е. не всегда есть возможность создать произвольный кластерный индекс.

См. подробное описание первичного, кластерного и вторичного (некластерного) индекса в книге «Fundamentals of Database Systems» (Ramez Elmasri, Shamkant Navathe), 6-е издание, глава 18.1 «Types of Single-Level Ordered Indexes».

74 «Clustered and Secondary Indexes» [https://dev.mysql.com/doc/refman/8.0/en/innodb-index-types.html]

По структуре хранения индексы (как и таблицы во многих методах доступа{33}) бывают разделёнными и неразделёнными:

Разделённый индекс (partitioned index[75]) — индекс, хранимый и обрабатываемый в виде отдельных частей (разделов, фрагментов) с целью повышения производительности.

Упрощённо: индекс, разделённый на несколько фрагментов.

Неразделённый индекс (non-partitioned index, index[76]) — индекс, хранимый и обрабатываемый как единая структура данных.

Упрощённо: «просто индекс» (без применения специальных команд по умолчанию все индексы создаются как неразделённые).

Идея разделения индексов очень близка к идее разделения таблиц — способа хранения данных таблицы в виде нескольких частей (что позволяет размещать их на отдельных физических накопителях и обрабатывать параллельно).

В зависимости от СУБД между разделением таблиц и индексов может быть следующая взаимосвязь:

- MySQL позволяет разделять таблицы и индексы только синхронно (т.е. нельзя построить неразделённый индекс на разделённой таблице или разделённый индекс на неразделённой таблице[77]);
- MS SQL Server позволяет использовать любые комбинации разделения индексов и таблиц (т.е. строить разделённые и неразделённые индексы как на разделённых, так и на неразделённых таблицах[78]);
- Oracle позволяет использовать любые комбинации разделения индексов и таблиц (т.е. строить разделённые и неразделённые индексы как на разделённых, так и на неразделённых таблицах[79]).

В подавляющем большинстве случаев (СУБД и методов доступа{33}) разделение кластерных индексов{110} и/или кластерных таблиц не поддерживается.

Поскольку особенности реализации разделённых индексов очень сильно зависят от выбранной СУБД (и даже её версии), мы ограничимся схематичным представлением нескольких вариантов разделения индексов и таблиц на примере того, как это реализовано в Oracle (рисунок 2.4.e).

Обязательно тщательно ознакомьтесь с документацией по вашей СУБД и/или выбранному методу доступа. Необдуманное использование разделённых индексов и/или таблиц может значительно снизить производительность СУБД и привести к иным нежелательным последствиям.

---

75 **Partitioned index** — an index broken into multiple pieces. («Understanding Partitioned Indexes in Oracle 11g», Richard Niemiec). [https://logicalread.com/oracle-11g-partitioned-indexes-mc02/]

76 **Non-partitioned index, index** — см. index{106}.

77 «Overview of Partitioning in MySQL» [https://dev.mysql.com/doc/refman/8.0/en/partitioning-overview.html]

78 «Create Partitioned Tables and Indexes» [https://technet.microsoft.com/en-us/library/ms188730(v=sql.130).aspx]

79 «Partitioning Concepts» [https://docs.oracle.com/cd/E11882_01/server.112/e25523/partition.htm]

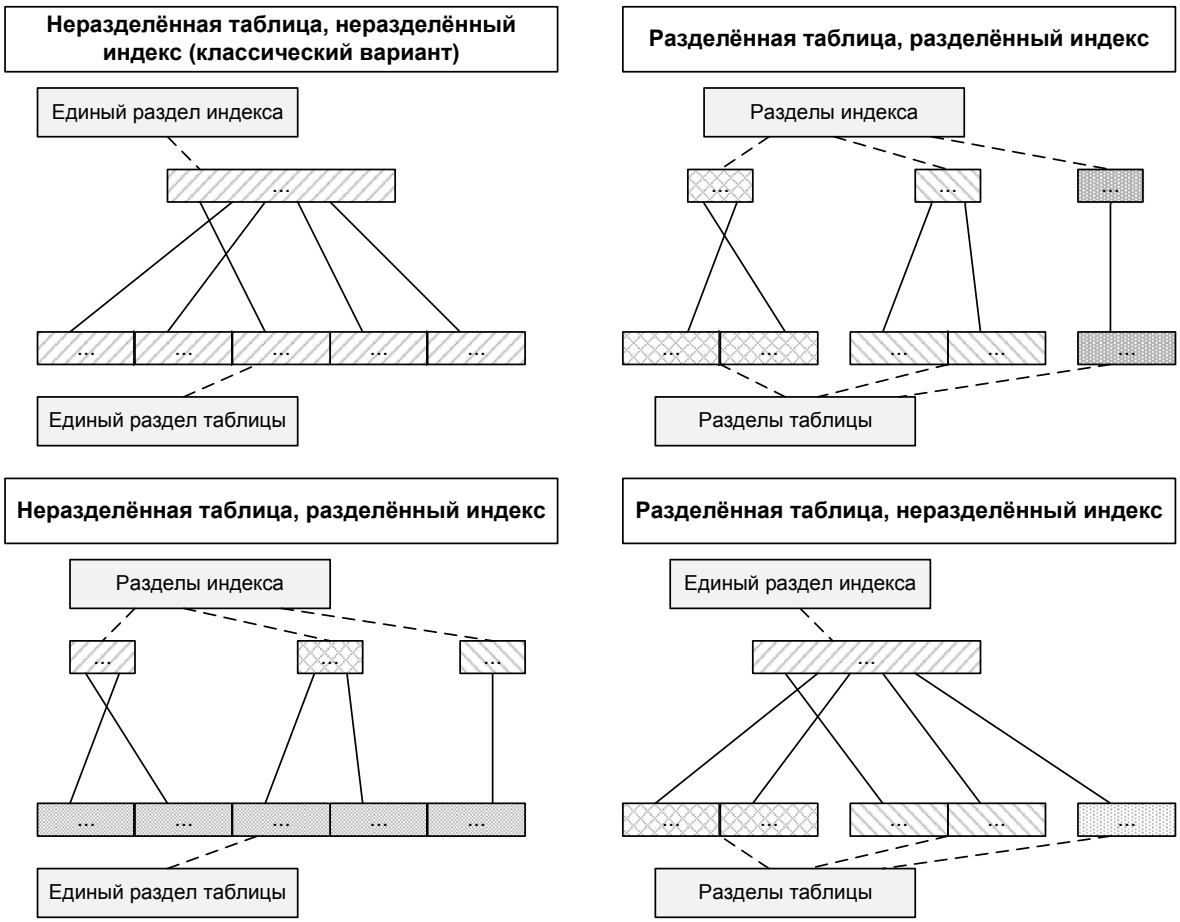


*Рисунок 2.4.e — **Схематичное представление вариантов использования разделённых индексов и таблиц***

И в завершение рассмотрения разделённых индексов схематично представим цель создания как самих разделённых индексов, так и разделённых таблиц (рисунок 2.4.f), состоящую в возможности распределения хранения данных по различным устройствам и параллельной обработки таких разделённых данных.

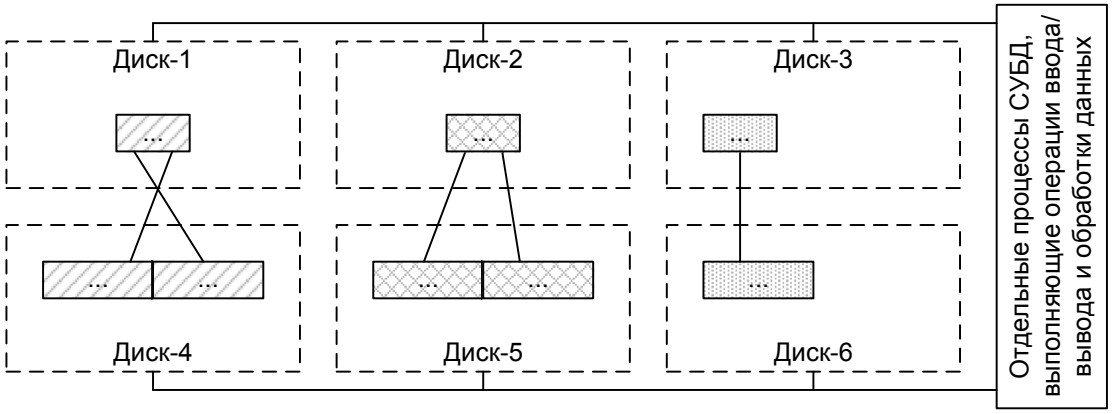


*Рисунок 2.4.f — **Схематичное представление цели создания разделённых индексов и таблиц***

Большинство современных СУБД поддерживает использование как разделённых, так и неразделённых индексов, но особенности их реализации и доступные возможности сильно отличаются в различных СУБД и методах доступа{33}.

По степени детализации индексы бывают плотными и неплотными:

Плотный индекс (dense index[80]) — индекс, содержащий указатель на расположение записи для каждого значения индексируемого поля.

Упрощённо: в индексе хранятся адреса каждой записи таблицы.

Неплотный индекс (sparse index[81]) — индекс, содержащий указатель на расположение блока записей для каждого значения (в случае их неуникальности) или группы значений (в случае их уникальности) индексируемого поля.

Упрощённо: в индексе хранятся адреса блоков (групп) записей.

Различие плотных и неплотных индексов проще всего пояснить в контексте ранее рассмотренных первичного{110}, кластерного{110} и некластерного{110} индексов.

Несмотря на то, что в теории первичные индексы могут быть неплотными (сейчас это будет рассмотрено), на практике почти у всех СУБД эти индексы являются плотными.

Проясним ситуацию с плотными и неплотными первичными индексами. По определению{110} такие индексы строятся на полях таблицы, содержащих уникальные значения. Но сам индекс (как показано на рисунке 2.4.g) может содержать указатели как на каждую отдельную запись, так и на блок записей (в таком случае конкретные значения индексируемого поля будут лежать в диапазоне «от текущего значения в индексе включительно, до следующего значения в индексе»).

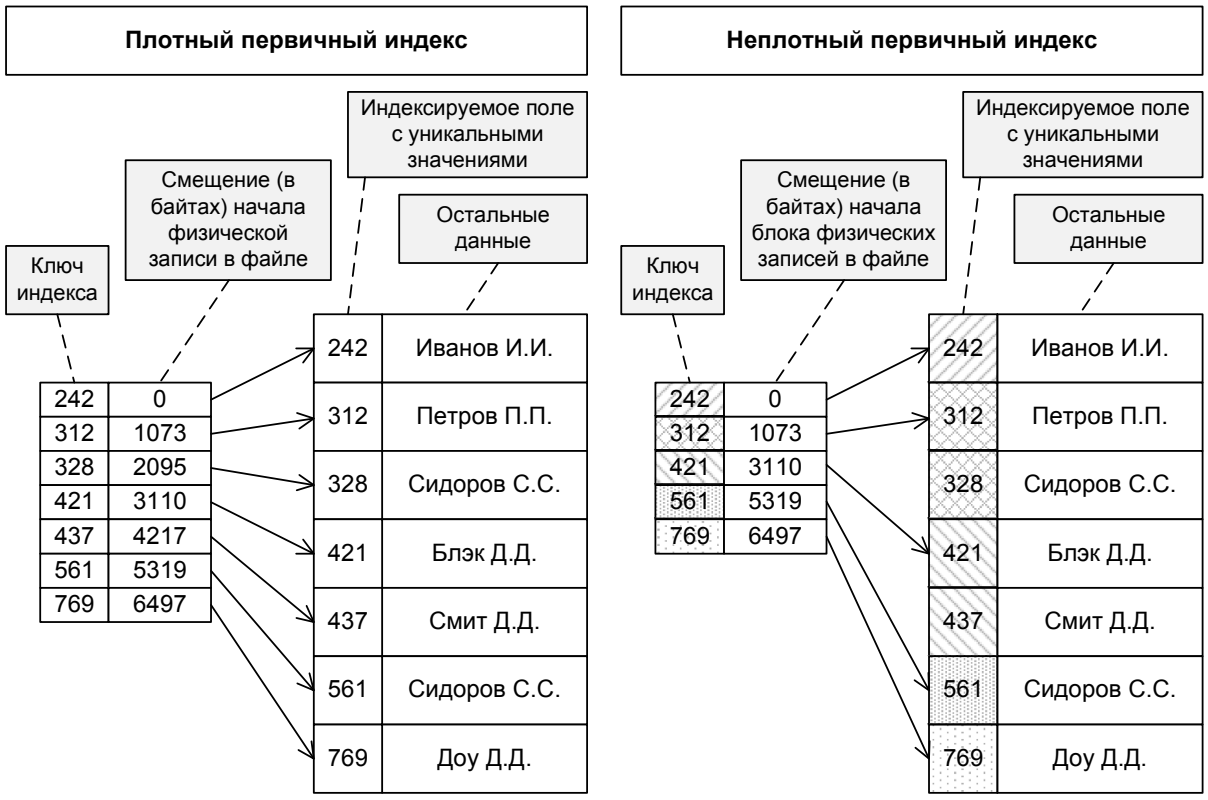


*Рисунок 2.4.g — **Плотный и неплотный первичный индекс***

---

[80] **Dense index** — an index that has an entry for every search key value (and hence every record) in the data file. («Fundamentals of Database Systems», Ramez Elmasri, Shamkant Navathe)

[81] **Sparse index** — an index that has entries for only some of the search values. («Fundamentals of Database Systems», Ramez Elmasri, Shamkant Navathe)

К преимуществам неплотных индексов относится их меньший размер и возможность более редкого обновления. Но это преимущество меркнет на фоне недостатка, состоящего в необходимости обращения к таблице для поиска конкретной записи, значение индексированного поля которой попало в тот или иной блок. Потому на практике разработчики СУБД предпочитают использовать плотные первичные индексы.

Стоит отметить, что кластерные (не первичные) индексы не так страдают от только что упомянутого недостатка: т.к. индексируемое поле в соответствующем блоке записей будет иметь одинаковые значения, для дальнейшего выполнения некоторых операций (например, извлечения остальных данных, не покрытых индексом) всегда придётся выполнять обращение к файлу таблицы, а для выполнения других операций (например, подсчёта количества уникальных значений) хранимой в индексе информации оказывается достаточно.

Из только что приведённых рассуждений должно быть очевидно, что некластерные индексы обязаны быть плотными, т.к. для них неактуально понятие «блока записей», обладающих упорядоченностью или неким общим признаком значений индексируемого поля.

Таким образом, кластерные индексы можно считать классическим случаем неплотных индексов, а некластерные — плотных (рисунок 2.4.h).

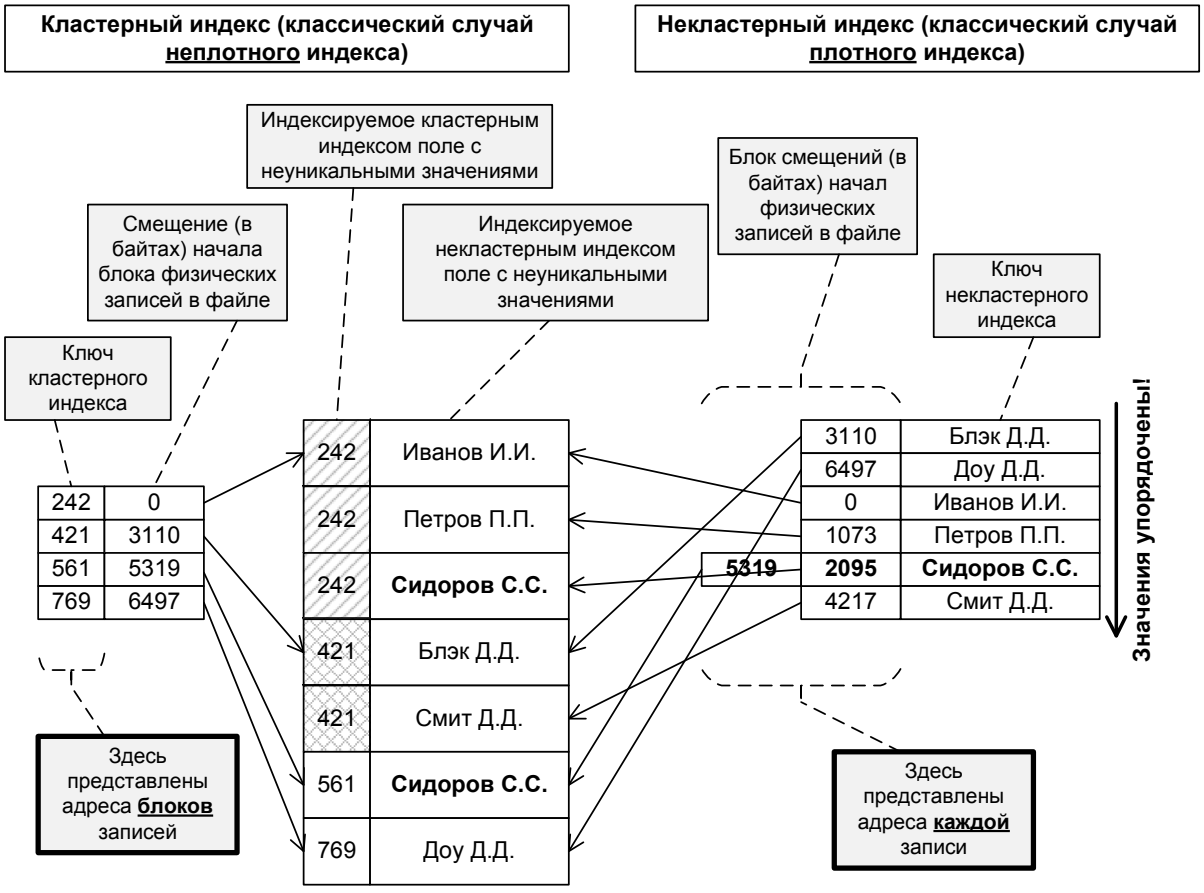


*Рисунок 2.4.h — **Кластерный и некластерный индексы как классические случаи неплотных и плотных индексов***

Большинство современных СУБД поддерживает использование как плотных, так и неплотных индексов, но в подавляющем большинстве случаев предпочтение отдаётся плотным индексам, и возможности выбора здесь сильно ограничены.

По базовой структуре индексы могут быть представлены очень большим количеством разновидностей, основными из которых являются:

Индекс на основе B-дерева (сбалансированного дерева) (B-tree[82] index) — индекс, структурно организованный с использованием B-дерева (сбалансированного дерева), оптимизированный для выполнения поиска на основе диапазонов и для операций с большими блоками данных. Допускает хранение части индекса во внешней памяти.

Упрощённо: одна из основных форм организации индексов в большинстве СУБД и методов доступа. (Буква «B» в названии индекса идёт от слова «balanced».)

Индекс на основе T-дерева (T-tree[83] index) — индекс, структурно организованный с использованием T-дерева (разновидности сбалансированного дерева, в котором вместо самих данных хранятся указатели на адреса данных в памяти), оптимизированный для выполнения операций в случае, когда и индекс и данные целиком находятся в оперативной памяти.

Упрощённо: индекс для СУБД и методов доступа, предполагающих хранение всего объёма обрабатываемых данных в оперативной памяти. (Буква «T» в названии индекса идёт от графической формы представления вершин T-дерева в статье, в которой оно было впервые представлено.)

Индекс на основе R-дерева (R-tree[84] index) — индекс, структурно организованный с использованием R-дерева (специальной формы представления географических и геометрических данных), оптимизированный для выполнения операций со специфическими типами данных, хранящих географические координаты или информацию о геометрических фигурах.

Упрощённо: индекс для ускорения обработки географических и геометрических данных (Буква «R» в названии индекса идёт от слова «rectangle».).

Индекс на основе хэш-таблицы (hash-table[85] index) — индекс, структурно организованный с использованием хэш-таблицы (специальной структуры для хранения пар ключ-значение), оптимизированный для выполнения поиска на основе строгого сравнения и обработки относительно редко изменяемых данных.

Упрощённо: одна из основных форм организации индексов в большинстве СУБД и методов доступа (наряду с индексами на основе B-деревьев).

---

82 **B-tree** — a self-balancing tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time. («Wikipedia») [https://en.wikipedia.org/wiki/B-tree]

83 **T-tree** — a balanced index tree data structure optimized for cases where both the index and the actual data are fully kept in memory, just as a B-tree is an index structure optimized for storage on block oriented secondary storage devices like hard disks. («Wikipedia») [https://en.wikipedia.org/wiki/T-tree]

84 **R-tree** — a tree data structure used for spatial access methods, i.e., for indexing multi-dimensional information such as geo-graphical coordinates, rectangles or polygons. «Wikipedia» [https://en.wikipedia.org/wiki/R-tree]

85 **Hash-table** — a data structure used to implement an associative array, a structure that can map keys to values. A hash table uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found. «Wikipedia» [https://en.wikipedia.org/wiki/Hash_table]

Индекс на основе битовой маски (bitmap[86] index) — индекс, структурно организованный с использованием битовой маски (специальной структуры для хранения информации о наличии в поле того или иного значения), оптимизированный для работы со столбцами, количество различных значений в которых относительно невелико.

Упрощённо: индекс хранит в очень компактной форме признак наличия в некоторой ячейке таблицы одного из значений, присутствующих в соответствующем столбце.

Описание самих структур данных и алгоритмов их обработки выходит далеко за рамки данной книги, но всё же приведём схематичное представление каждого из перечисленных индексов (рисунки 2.4.i, 2.4.j, 2.4.k, 2.4.l, 2.4.m), после чего перейдём к рассмотрению областей их применения.

[86] **Bitmap** — an array data structure that compactly stores bits. It can be used to implement a simple set data structure. A bitmap is effective at exploiting bit-level parallelism in hardware to perform operations quickly. «Wikipedia» [https://en.wikipedia.org/wiki/Bit_array]

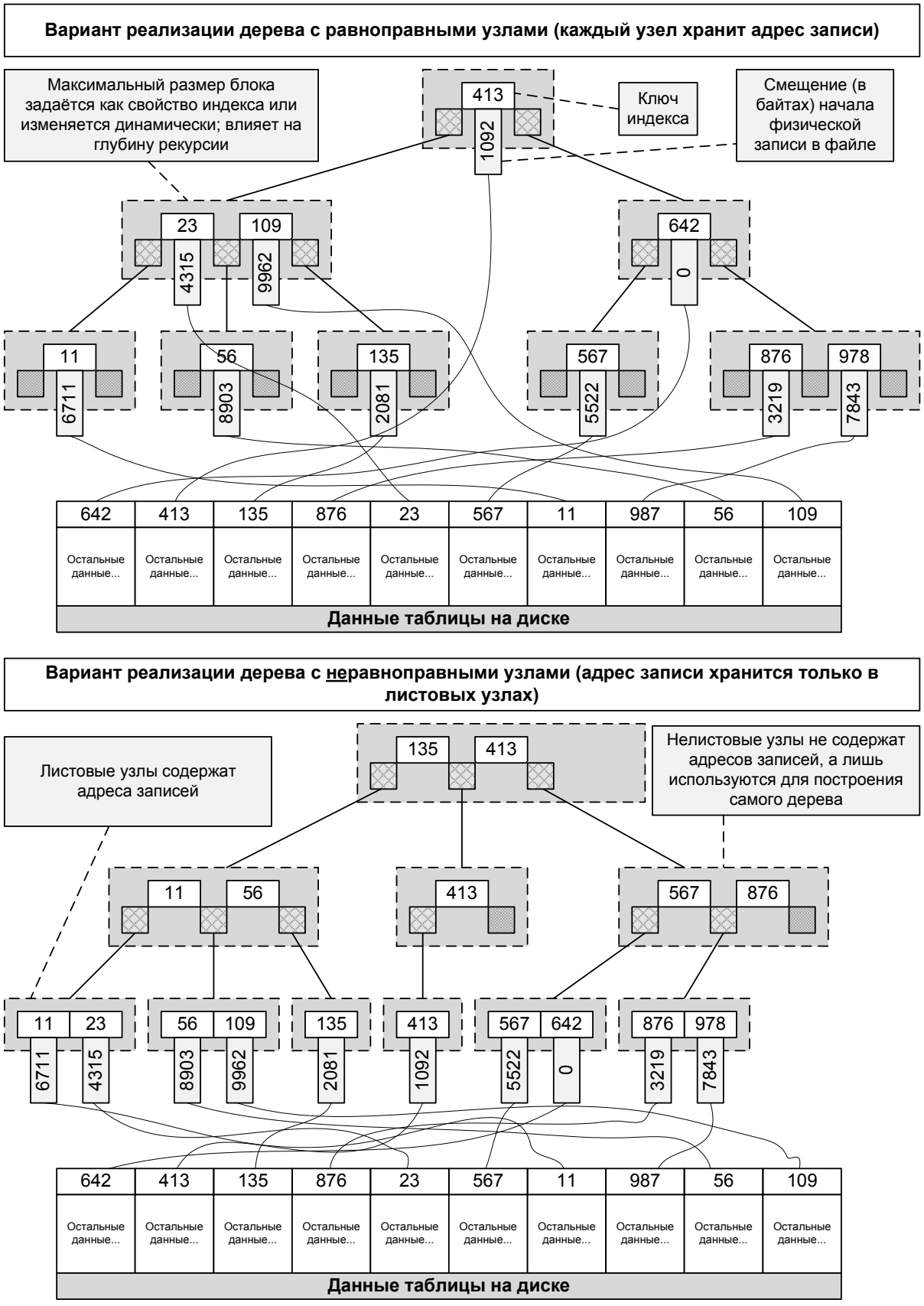


*Рисунок 2.4.i — **Схематичное представление индекса на основе B-дерева***

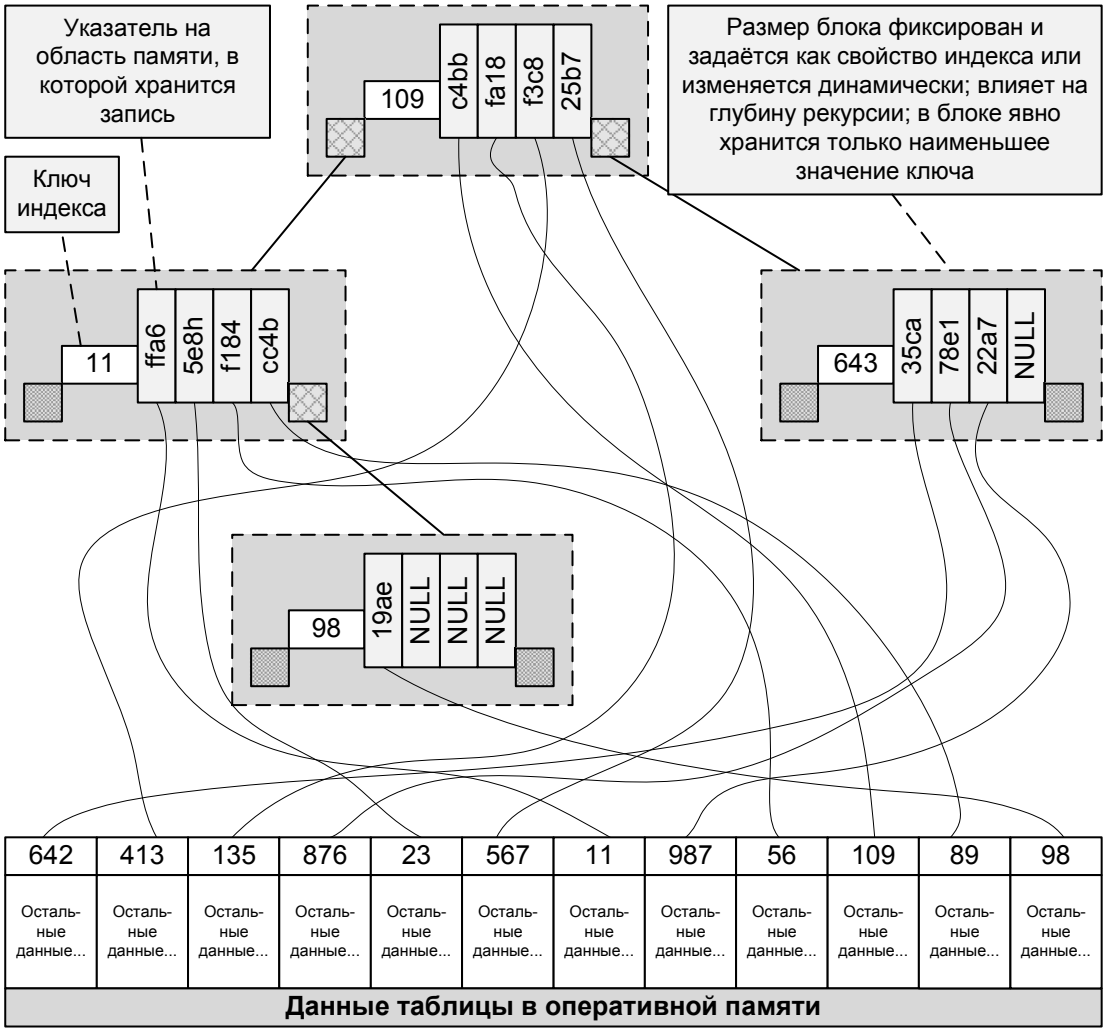


*Рисунок 2.4.j — **Схематичное представление индекса на основе T-дерева***

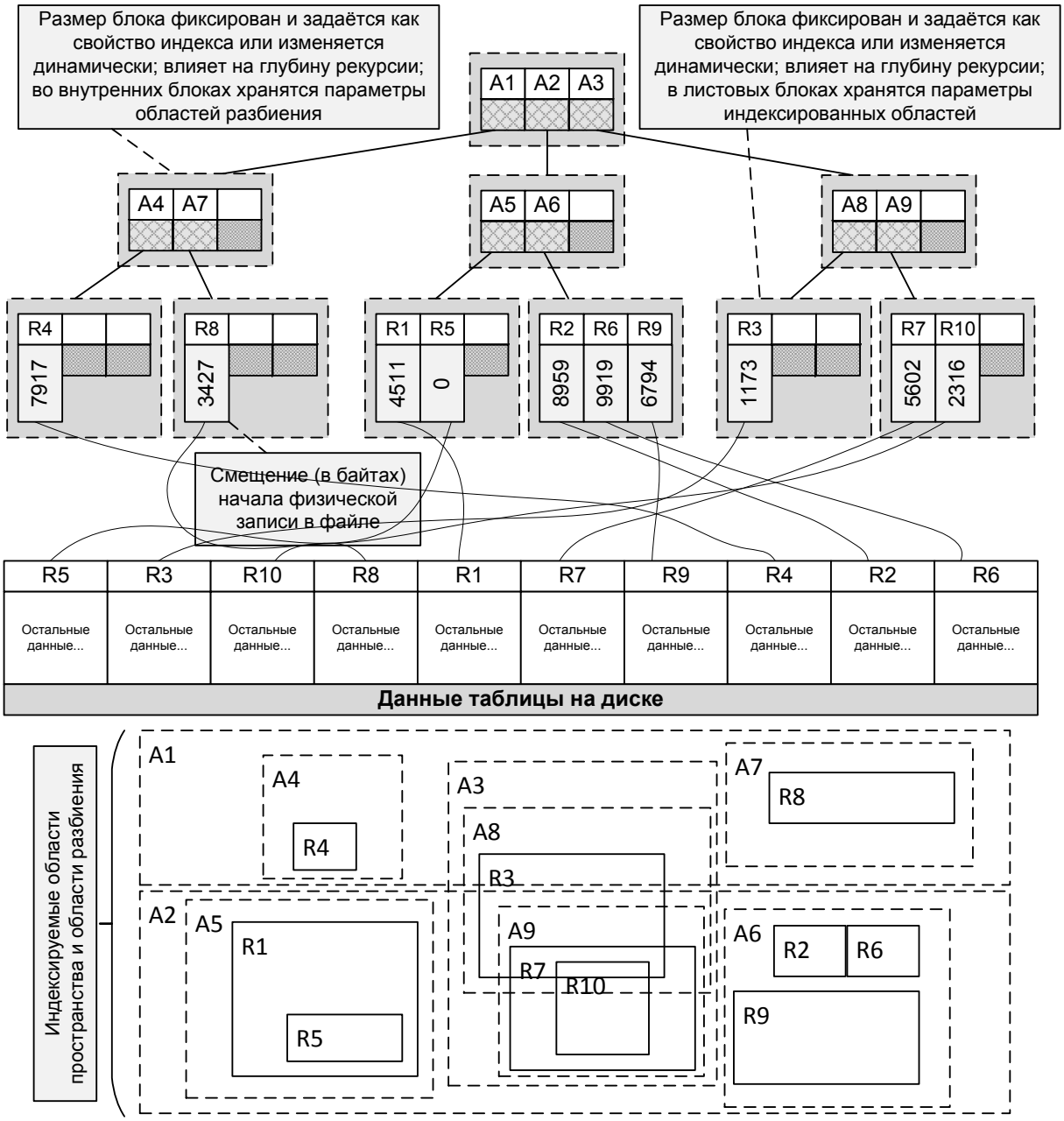


Рисунок 2.4.k — ***Схематичное представление индекса на основе R-дерева***

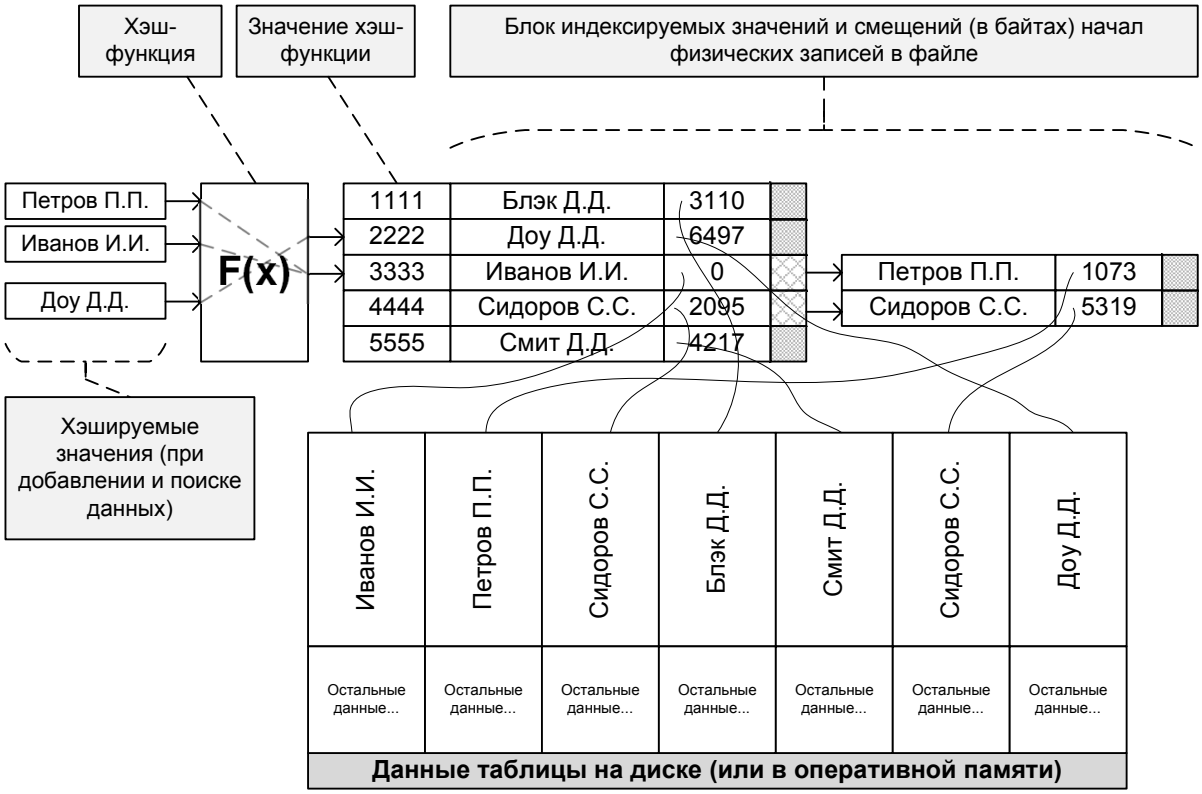


*Рисунок 2.4.l — **Схематичное представление индекса на основе хэш-таблицы***

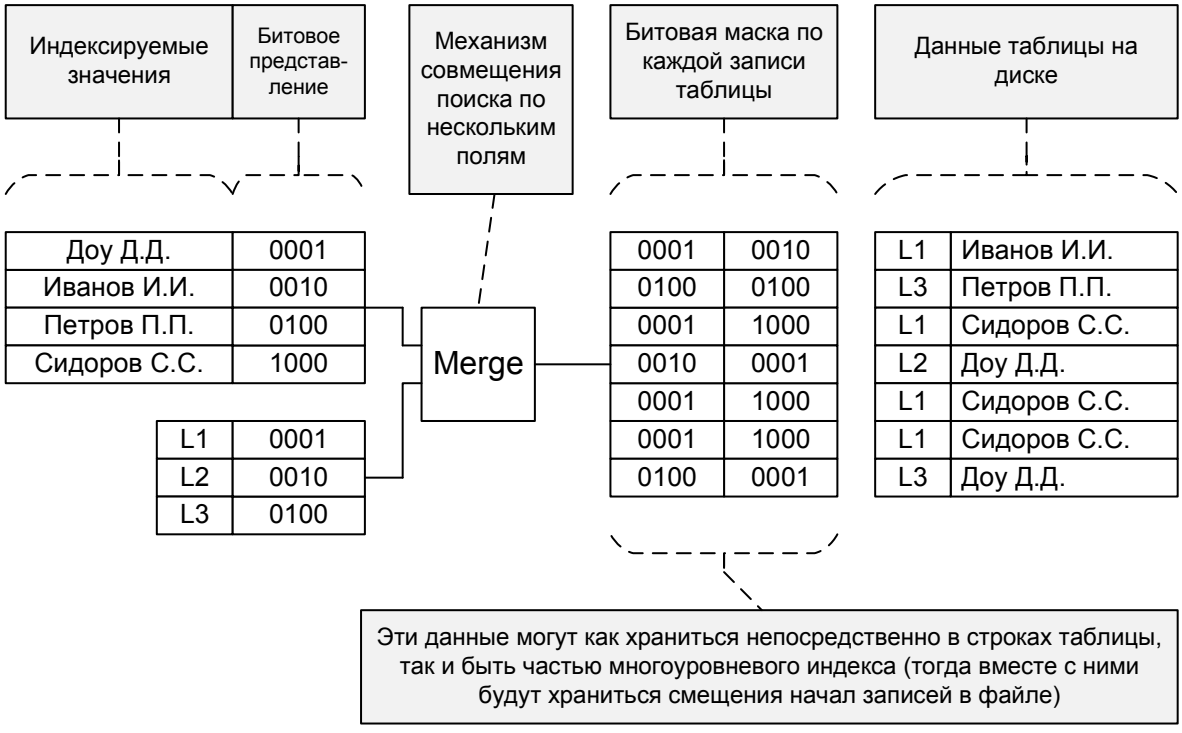


*Рисунок 2.4.m — **Схематичное представление индекса на основе битовой маски***

Теперь рассмотрим основные области применения каждой разновидности представленных в данной классификации индексов. Обратите внимание: здесь речь не идёт о преимуществах и недостатках, т.к. они очень сильно зависят от конкретной ситуации.

Индекс на основе…		**B-дерева**	**T-дерева**	**R-дерева**	**Хэш-таблицы**	**Битовой маски**
Основная область применения…		Типичная форма реализации большинства индексов, эффективны при операциях =, >, ≥, <, ≤, могут подгружаться в память частями	В основном — для т.н. in-memory баз данных (когда все данные удерживаются в оперативной памяти)	Пространственные и геометрические данные	Вторая по частоте использования (после B-деревьев) форма реализации индексов, эффективны при операциях = и !=	Столбцы с малой мощностью (малым количеством различающихся значений)
Бывает…	Простым	Да	Да	Да	Да	Да
	Составным	Да	Да	Теоретически	Да	Да
	Уникальным	Да	Да	Теоретически	Да	Да
	Неуникальным	Да	Да	Да	Да	Да
	Кластерным	Да	Теоретически	Теоретически	Теоретически	Да
	Некластерным	Да	Да	Да	Да	Да
	Разделённым	Да	Теоретически	Теоретически	Теоретически	Теоретически
	Неразделённым	Да	Да	Да	Да	Да
	Плотным	Да	Да	Да	Да	Да
	Неплотным	Да	Теоретически	Теоретически	Теоретически	Да
	Одноуровневым	Крайне редко	Крайне редко	Теоретически	Да	Да
	Многоуровневым	Да	Да	Да	Теоретически	Теоретически
	Покрывающим	Да	Да	Да	Да	Да
	Непокрывающим	Да	Да	Да	Да	Да

Здесь варианты «крайне редко» и «теоретически» означают, что в принципе создать такой индекс можно, но на практике он не применяется или используется в очень редких случаях либо в силу низкой эффективности, либо в силу высокой сложности реализации.

В контексте различных СУБД ситуация такова: подавляющее большинство СУБД поддерживают индексы на основе сбалансированных деревьев и хэш-таблиц, многие — на основе R-деревьев, а вот индексы на основе T-деревьев и битовых масок распространены меньше всего.

По сложности иерархии индексы делятся на одноуровневые и многоуровневые, причём обе эти разновидности уже встречались в данной главе, пусть и оставаясь не названными явно.

Одноуровневый индекс (single-level index[87]) — индекс, структура которого является плоской, т.е. содержит ровно один уровень.

Упрощённо: индекс без иерархии.

Многоуровневый индекс (multi-level index[88]) — индекс, структура которого является иерархической, т.е. содержит два и более уровня. Такие уровни могут содержать однотипную информацию или различаться по своему назначению (как правило, листовые узлы будут отличаться от нелистовых, в то время как все нелистовые вне зависимости от их уровня выполняют одинаковую роль).

Упрощённо: индекс с иерархией (как правило, на основе дерева).

Типичными представителями одноуровневых индексов являются:
- первичный индекс{110};
- кластерный индекс{110};
- индекс на основе хэш-таблицы{117};
- индекс на основе битовой маски{118}.

Типичными представителями многоуровневых индексов являются:
- индекс на основе B-дерева{117};
- индекс на основе T-дерева{117};
- индекс на основе R-дерева{117}.

Стоит отметить, что теоретически любой многоуровневый индекс можно свести к одноуровневому и наоборот, но на практике индексы потому и бывают этих двух видов, что такая организация оказывается наиболее эффективной для решения возложенных на тот или иной индекс задач.

Большинство современных СУБД поддерживает использование как одноуровневых индексов (как минимум кластерных и/или первичных), так и многоуровневых (как правило, на основе B-деревьев).

По соотношению с SQL-запросом индексы бывают покрывающими и непокрывающими — и данное разделение во всей рассматриваемой нами классификации является единственным условным, т.е. один и тот же индекс может быть как покрывающим, так и не покрывающим, потому что здесь есть зависимость не только от самого индекса, но и от SQL-запроса.

Покрывающий индекс (covering index[89]) — индекс, содержащий в явном виде внутри себя информацию, достаточную для выполнения SQL-запроса без обращения к данным, хранящимся вне этого индекса (в самой таблице).

Упрощённо: индекс, информации в котором достаточно для выполнения SQL-запроса без обращения к данным в таблице.

---

87 **Single-level index** — an index based on ordered file. («Fundamentals of Database Systems», Ramez Elmasri, Shamkant Navathe)

88 **Multi-level index** — an index based on tree data structure. («Fundamentals of Database Systems», Ramez Elmasri, Shamkant Navathe)

89 **Covering index** — an index that contains all information required to resolve the query is known as a "Covering Index"; it completely covers the query («Using Covering Indexes to Improve Query Performance», Joe Webb). [https://www.simple-talk.com/sql/learn-sql-server/using-covering-indexes-to-improve-query-performance/]

Непокрывающий индекс (non-covering index[90]) — индекс, позволяющий лишь ускорить нахождение нужной информации, в то время как сама искомая информация внутри индекса не содержится или содержится не полностью.

*Упрощённо: «просто индекс», информации в котором **не**достаточно для выполнения SQL-запроса без обращения к данным в таблице.*

Продемонстрируем суть покрытия индексом запроса на примере. Допустим, у нас есть следующая таблица `users`:

Первичный ключ, кластерный индекс	Составной некластерный индекс {u_email, u_status}		На этих столбцах индексов нет	
u_id	**u_email**	**u_status**	**u_login**	**u_name**
1	ivanov@mail.ru	Active	ivanov	Иванов И.И.
2	petrov@mail.ru	Active	petrov	Петров П.П.
3	sidorov@mail.ru	Locked	sidorov	Сидоров С.С.
4	smith@gmail.com	Active	smith	Смит С.
5	dow@yahoo.com	Locked	dow	Доу Д.

Рассмотрение запросов начнём с достаточно спорного примера:

MySQL **Первый пример покрытого индексом запроса**

```
SELECT COUNT(`u_id`)
FROM   `users`
WHERE  `u_id` >= 2
       AND `u_id` <= 10
```

С одной стороны, да — индекс на поле `u_id` содержит всю необходимую информацию для выполнения запроса, и к данным в таблице можно не обращаться. Т.е. данный индекс является покрывающим для данного запроса.

С другой стороны, в некоторых СУБД и методах доступа{33}, кластерный индекс — это, физически, ничто иное как сама таблица (с указанием того факта, что она упорядочена по некоторому полю). Тогда формально здесь индекс можно не считать покрывающим, хотя скорость выполнения запроса всё равно будет выше, чем если бы этого индекса не было.

Рассмотрим более классический пример:

MySQL **Второй пример покрытого индексом запроса**

```
SELECT `u_status`
FROM   `users`
WHERE  `u_email` = 'ivanov@mail.ru'
```

Для данного запроса индекс `{u_email, u_status}` является покрывающим, т.к. содержит в себе значения всех используемых в запросе полей (по первому полю `u_email` в составе составного индекса СУБД может производить поиск столь же быстро, как и по всему индексу целиком, а значение извлекаемого поля `u_status` также содержится в индексе).

90 **Non-covering index** — см. index{106}.

И, наконец, приведём пример непокрытого индексом запроса:

MySQL **Пример непокрытого индексом запроса**

```
1   SELECT `u_login`
2   FROM   `users`
3   WHERE  `u_email` = 'ivamov@mail.ru'
4      AND `u_status` = 'Active'
```

Для данного запроса индекс **{u_email, u_status}** не является покрывающим, т.к. несмотря на то, что с его помощью СУБД очень быстро выполнит поиск нужной записи, значение поля **u_login** придётся извлекать из данных самой таблицы.

Очевидно, что вопрос о том, какие СУБД поддерживают покрывающие и непокрывающие индексы, является бессмысленным: любой индекс может быть как покрывающим, так и непокрывающим в зависимости от выполняемого SQL-запроса.

По специфическим функциям индексы не столько разделяются на группы, сколько получают свои особые названия в тех или иных СУБД. Также отметим, что с каждым годом появляются всё новые и новые специфические индексы, потому представленный ниже список, возможно, уже является неполным.

Колоночный индекс (columnstore index[91]) — набор решений для хранения и обработки данных таким образом, что базовой единицей является не ряд (как в классическом случае), а столбец.

Упрощённо: индекс для быстрой обработки отдельных столбцов.

Данный вид индекса был представлен в MS SQL Server 2012, пришёл туда из области хранилищ данных и не зря называется «набором решений». Фактически, это не только и не столько индекс, сколько способ организации данных в таблице (если этот индекс является кластерным) или способ очень быстрого доступа к отдельным столбцам таблицы.

Структуру такого индекса продемонстрируем графически в сравнении с обычным индексом (рисунок 2.4.n).

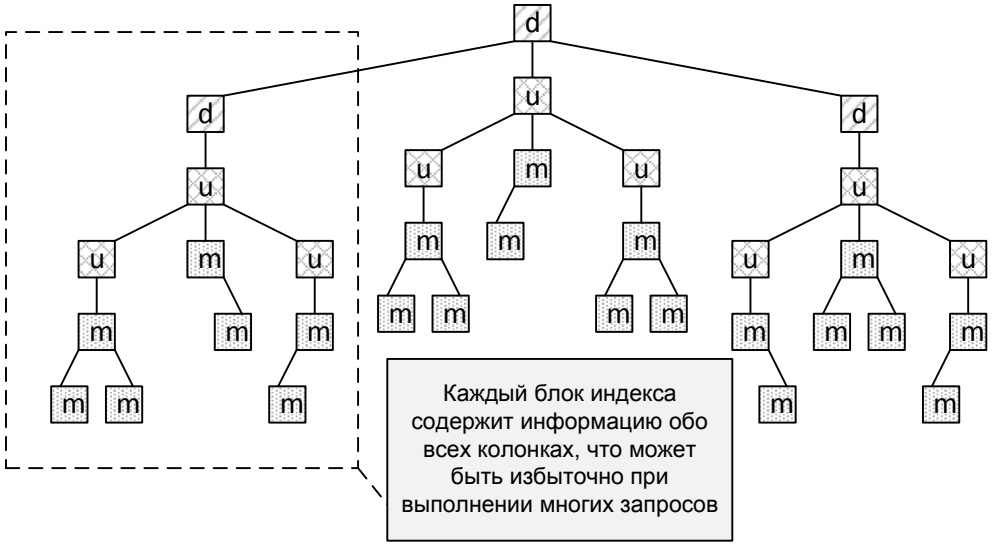
Стоит понимать, что колоночный индекс — не «магия», и не позволяет улучшить любой запрос (более того, в «типичной базе данных», где поисковые запросы превалируют над аналитическими, такой индекс будет скорее вредным, чем полезным).

Но в представленном на рисунке 2.4.n примере запросы вида «показать количество платежей по датам», «показать среднее арифметическое и медиану платежей», «показать минимальный и максимальный платежи» и им подобные будут быстрее работать с использованием колоночного индекса.

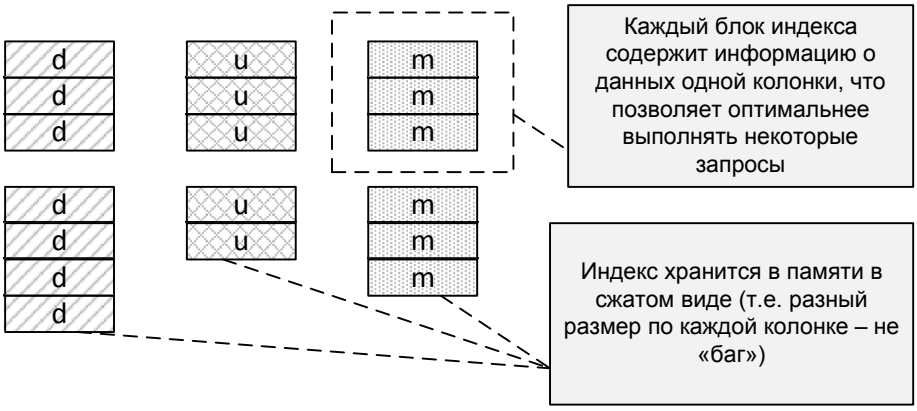
---

[91] **Columnstore index** — a technology for storing, retrieving and managing data by using a columnar data format, called a columnstore («Columnstore Indexes Guide»). [https://msdn.microsoft.com/en-us/library/gg492088(v=sql.130).aspx]

Классический индекс на основе дерева
(в данном случае индекс построен на полях {p_date, p_user, p_money})



Колоночный индекс на тех же полях {p_date, p_user, p_money}



Пример типичных данных, для которых построение колоночного индекса может оказаться эффективным при выполнении многих аналитических запросов

p_id	p_user	p_money	p_date	...
1	234	34556	2016-02-12	...
2	89	565	2016-03-18	...
3	34	341235	2015-09-02	...
4	2342	24234	2017-02-14	...
...	...	...	...	...
34526256	34235	21321	2016-12-19	...

*Рисунок 2.4.n — **Схематичное представление структуры колоночного индекса***

Индекс со включёнными столбцами (index with included columns[92]) — некластерный{110} индекс, содержащий в своих листовых узлах информацию из дополнительного поля, которое не используется при построении самого индекса.

Упрощённо: индекс, позволяющий покрывать{124} больше запросов за счёт содержания в себе дополнительных данных.

Поясним различие между составным{108} индексом и индексом со включёнными столбцами. В составном индексе все поля используются при построении индекса, а в индексе со включёнными столбцами данные из этих включённых столбцов просто хранятся и не влияют на логику построения индекса (рисунок 2.4.о).

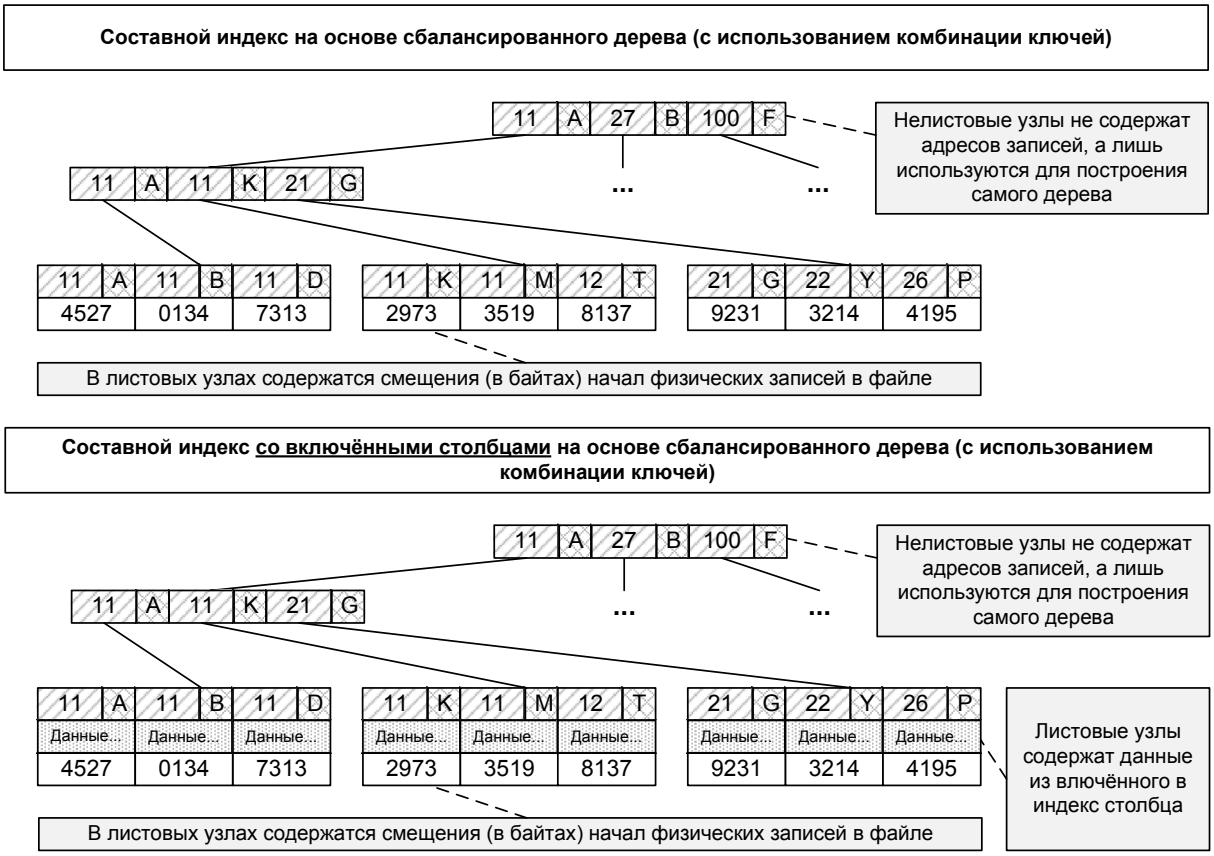


*Рисунок 2.4.о — **Структура составного индекса и индекса со включёнными столбцами***

Наличие в листовых узлах данных из включённого столбца позволяет не обращаться к таблице при выполнении запроса, в котором нас интересуют эти данные из включённого столбца, что позволяет такому индексу покрывать запросы, которые не были бы покрыты «обычным индексом», и в то же время избежать перерасхода памяти (за счёт того, что данные из включённого столбца не учитываются при построении индекса и не хранятся в нелистовых узлах).

Очевидно, что такой индекс не может быть использован для поиска по значениям включённого столбца — он лишь позволяет быстро извлекать эти значения, но сам поиск должен выполняться по значениям основных полей индекса.

Такой индекс используется в MS SQL Server, но в других СУБД распространения не получил.

---

92 **Index with included columns** — a nonclustered indexes that includes nonkey columns to cover more queries («Create Indexes with Included Columns»). [https://msdn.microsoft.com/en-us/library/ms190806.aspx]

Индекс на вычисляемых столбцах (index on computed columns[93]) — индекс, построенный на значениях виртуальных столбцов, данные которых могут не храниться физически в базе данных.

Упрощённо: индекс на столбце, значение которого вычисляет сама СУБД.

Индекс на значениях функций (function-based index[94]) — индекс, построенный на результатах применения к хранящимся данным функции (встроенной или пользовательской).

Упрощённо: индекс на результатах применения к данным некоторой функции (значение функции вычисляет сама СУБД).

Эти две разновидности индексов не зря рассматриваются вместе, т.к. они, по сути, являются решением одной и той же задачи, но первый (на вычисляемых столбцах) используется в MS SQL Server, а второй (на значениях функций) используется в Oracle.

Начиная с 11-й версии, Oracle также поддерживает вычисляемые столбцы, но несмотря на одинаковое название, реализация (и возможности) этого механизма в MS SQL Server и Oracle имеют серьёзные различия. Обязательно внимательно изучите соответствующие разделы документации.

Задача, решаемая с использованием этих индексов, может быть сформулирована как ускорение доступа к данным, значения которых вычисляются на уровне СУБД, а не передаются и сохраняются в базу данных явно (как это происходит в подавляющем большинстве случаев). На таких данных невозможно (или крайне сложно и неэффективно) строить «классические» индексы, именно поэтому некоторые СУБД предлагают рассматриваемое здесь решение.

Поясним его графически (рисунок 2.4.р). Допустим, что (для MS SQL Server) в каких-то операциях нам необходимо очень быстро искать людей по их инициалам (для этого создаётся вычисляемый столбец `u_initials`), а также (для Oracle) что для других операций нам необходимо выполнять поиск по полному ФИО в верхнем регистре, где данные идут в порядке имя, отчество, фамилия (для этого создаётся функция, значения которой индексируются).

Технически такие индексы могут реализовываться почти любым из ранее рассмотренных способов, но чаще всего это будут плотные{115} некластерные{110} индексы на основе B-деревьев{117}.

---

93 **Index on computed columns** — an index that is built on virtual column that is not physically stored in the table, unless the column is marked persisted; a computed column expression can use data from other columns to calculate a value for the column to which it belongs («Indexes on Computed Columns») [https://msdn.microsoft.com/en-us/library/ms189292.aspx]

94 **Function-based index** — rather than indexing a column, you index the function on that column, storing the product of the function, not the original column data; when a query is passed to the server that could benefit from that index, the query is rewritten to allow the index to be used («Oracle Function-Based Indexes»). [https://oracle-base.com/articles/8i/function-based-indexes]

*Рисунок 2.4.p — **Индексы на вычисляемых столбцах и значениях функции***

Как и было упомянуто выше, индексы двух данных типов актуальны в первую очередь для MS SQL Server и Oracle, но есть основания полагать, что и другие СУБД в обозримом будущем представят аналогичные решения.

Индекс с фильтром (filtered index[95]) — индекс, учитывающий из всего множества значений индексируемого столбца лишь (небольшую) часть, удовлетворяющую указанному при создании индекса условию.

Упрощённо: индекс на части значений столбца.

На практике часто возникает потребность в выполнении операций, затрагивающих лишь отдельную (как правило — небольшую) часть данных, удовлетворяющую некоторым заранее определённым критериям (например: рассылка сообщений только активным пользователям, отображение списка только незавершённых дел, ручная проверка только платежей больше некоторой суммы и т.д.)

В таких ситуациях эффективным является использование индексов с фильтром, которые содержат в себе информацию только о записях, удовлетворяющих заданному критерию. Такие индексы не только позволяют быстрее выполнять поиск, но и занимают меньше памяти, и быстрее обновляются.

Схематично принцип работы индексов с фильтром показан на рисунке 2.4.q (допустим, для некоторых операций со списком пользователей нам постоянно нужны только те из них, кого зовут «Петров П.П.» или «Сидоров С.С.», а любые другие не нужны).

---

[95] **Index on computed columns** — an index that is built on virtual column that is not physically stored in the table, unless the column is marked persisted; a computed column expression can use data from other columns to calculate a value for the column to which it belongs («Indexes on Computed Columns»). [https://msdn.microsoft.com/en-us/library/ms189292.aspx]

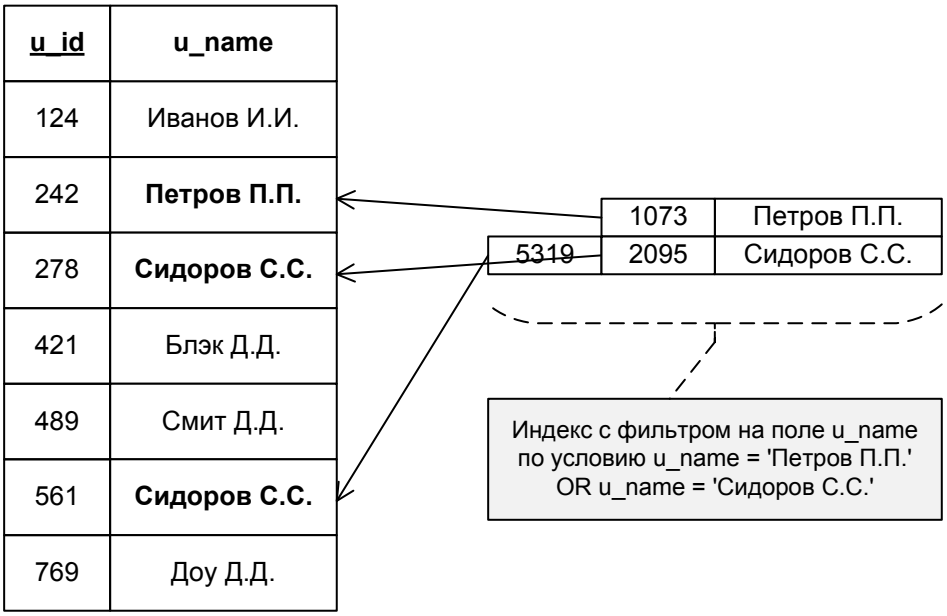


*Рисунок 2.4.q — **Принцип работы индексов с фильтром***

Индексы с фильтром явным образом поддерживаются в MS SQL Sever, а в некоторых других СУБД могут быть эмулированы через условные конструкции[96].

Пространственный индекс (spatial index[97]) — индекс, оптимизированный для ускорения операций поиска значений в столбцах, хранящих данные пространственного или геометрического типа.

Упрощённо: индекс на столбцах с пространственным или геометрическим типом данных.

Такие индексы могут строиться с использованием R-деревьев{117} или на основе B-деревьев{117} и многоуровневого разбиения пространства на однотипные вложенные области (такое решение применяется в MS SQL Server и показано на рисунке 2.4.r — количество уровней может варьироваться в разных реализациях, но в MS SQL Server оно равно четырём).

Область применения пространственных индексов ограничена (как следует из их определения) пространственными и геометрическими данными, но здесь они позволяют весьма эффективно оптимизировать производительность многих операций.

---

[96] Эмуляция индексов с фильтром в Oracle: https://asktom.oracle.com/pls/apex/f?p=100:11:0::::P11_QUESTION_ID:4092298900346548432

[97] **Spatial index** — a type of extended index that allows you to index a spatial column (a table column that contains data of a spatial data type, such as geometry or geography) («Spatial Indexes Overview»). [https://msdn.microsoft.com/en-us/library/bb895265.aspx]

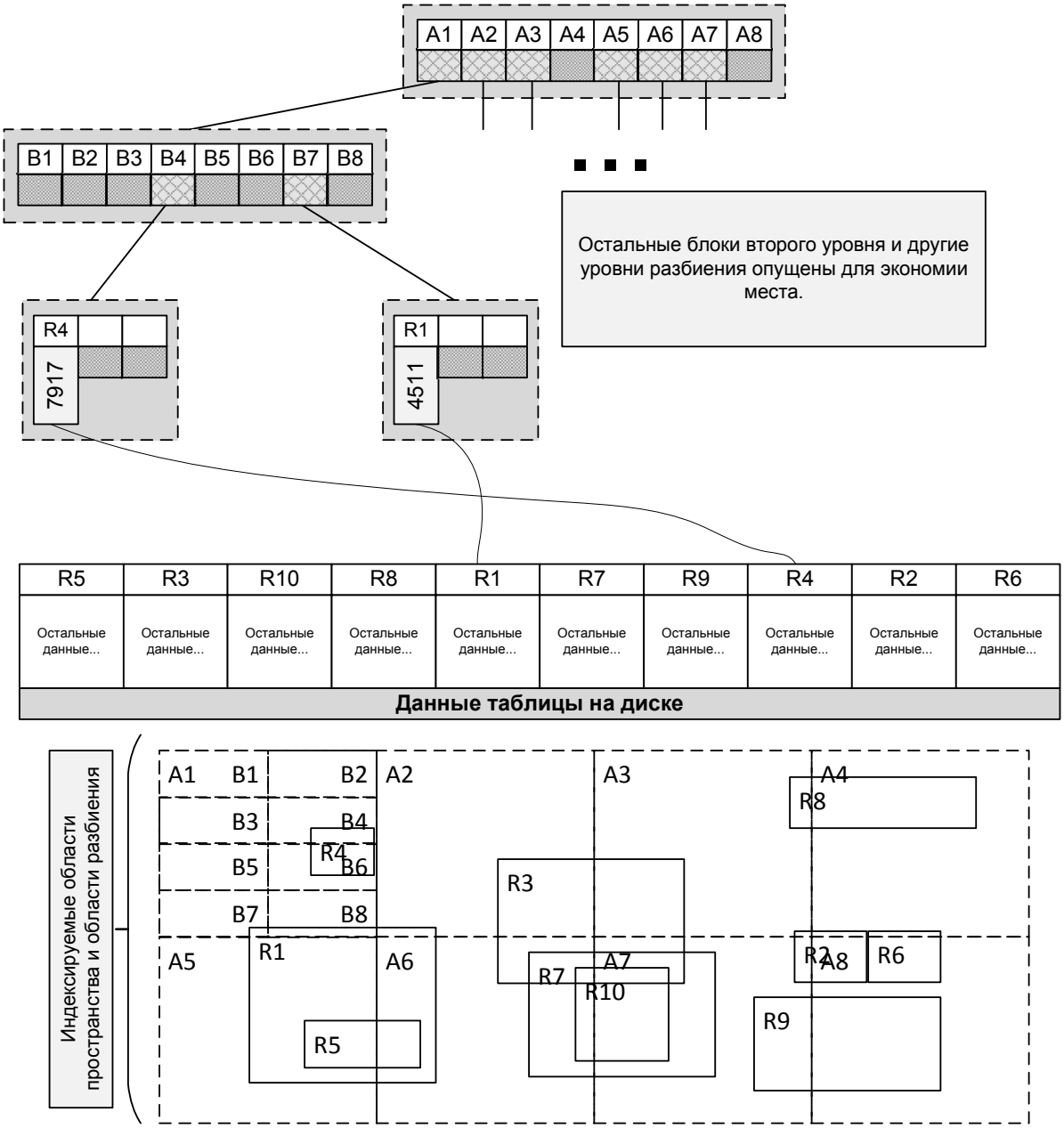


Рисунок 2.4.r — ***Один из вариантов реализации пространственного индекса (другой вариант на основе R-дерева был рассмотрен ранее***{117}***)***

В настоящий момент пространственные индексы поддерживаются MySQL, MS SQL Server, Oracle и другими СУБД, поддерживающими пространственные и геометрические типы данных.

Полнотекстовый индекс (full text index[98]) — индекс, оптимизированный для ускорения операций поиска вхождений подстрок в значения текстовых полей.

Упрощённо: индекс для поиска текста в тексте.

Ранее рассмотренные в данной главе «классические» индексы направлены на строгое сравнение искомых и хранимых значений или на проверку вложенности и пересечения областей (актуально для пространственных{131} индексов). Все эти подходы оказываются бесполезными при решении одной специфической, но очень востребованной задачи — поиска текста в тексте.

Самый частый случай возникновения этой задачи вам точно знаком — это поиск информации в Интернет с помощью поисковых сервисов. Конечно, используемые там алгоритмы и способы их реализации отличаются от того, что мы сейчас рассмотрим, но концепция остаётся неизменной: есть много текстов, из которых нужно выбрать те, что содержат искомые слова или искомую фразу.

В случае реализации собственного механизма поиска (для блога, интернет-магазина, форума, новостного сайта) одним из возможных решений может стать использование полнотекстовых индексов.

Несмотря на колоссальные различия реализации полнотекстовых индексов в разных СУБД, общий принцип их работы (проиллюстрированный на рисунке 2.4.s) схож: анализируемый текст разбивается на слова, и затем для каждого слова хранится список записей, содержащих в проиндексированном поле текст с этим словом. Часто в дополнение сохраняется позиция (позиции) вхождения этого слова в текст, чтобы иметь возможность выполнения поиска с учётом расстояния между словами.

Несмотря на всю красоту и мощь полнотекстовых индексов, стоит понимать, что они обладают широким спектром ограничений, а их использование требует точного понимания того, что вы делаете (для осознания масштаба сложности рекомендуется ознакомиться с документацией по MS SQL Server, где полнотекстовым индексам посвящено 28 *разделов*). Иными словами: чудес ожидать не стоит, но некоторые запросы можно ощутимо ускорить.

Итак, переходим к графическому представлению суть полнотекстовых индексов (рисунок 2.4.s). Допустим, у нас есть таблица, хранящая историю переписки в некоем корпоративном мессенджере. Для ускорения поиска по сообщениям мы построили полнотекстовый индекс на соответствующем поле (и указали в качестве «стоп-слов» слова «и», «где», «когда», «быть», «было», «будет», «спасибо».

Физически полнотекстовый индекс может быть реализован на основе B-дерева{117}, T-дерева{117}, хэш-таблицы{117} или иного алгоритма. Для простоты будем считать, что в нашем случае используется хэш-таблица{117}.

---

98 **Full text index** — indexes that is created on text-based columns to help speed up queries on data contained within those columns, omitting any words that are defined as stopwords («InnoDB FULLTEXT Indexes»). [https://dev.mysql.com/doc/refman/8.0/en/innodb-fulltext-index.html]

*Рисунок 2.4.s — **Схематичное представление полнотекстового индекса***

Полнотекстовые индексы реализованы практически во всех современных СУБД, но механизмы их реализации, возможности и ограничения в каждой СУБД свои — потому остаётся ещё раз повторить настоятельный совет: внимательно читайте документацию.

Доменный индекс (domain index[99]) — индекс, используемый для работы со специфическими данными в конкретной предметной области; взаимодействие с таким индексом реализуется через пользовательские подпрограммы (в отличие от «обычных» индексов, управление которыми уже реализовано в самой СУБД).

Упрощённо: индекс, логикой работы которого вы управляете сами.

Это весьма специфический вид индексов, который используется в Oracle. Для начала стоит отметить, что сама по себе СУБД Oracle реализует пространственные и полнотекстовые индексы с использованием механизма доменных индексов. Осознание этого факта позволяет лучше понять часть определения, в которой сказано про «работу со специфическими данными в конкретной предметной области» — согласитесь, что поиск вложенных и пересекающихся географических областей вполне подходит под такую специфику.

Технически такой индекс может быть реализован на основе битовой маски{118}, хэш-таблицы{117} или B-дерева{117}. А его особенное поведение должно быть описано вами как авторами индекса (и реализовано с использованием интерфейса `ODCIIndex`).

Пример кода такой реализации можно найти в документации[100], а мы ограничимся перечислением нескольких задач, которые можно решить с помощью доменного индекса:

- поиск в тексте с учётом количества разных форм каждого из составляющего текст слов;
- обработка временных интервалов с учётом их длительности и пересечения;
- индексирование сериализованных данных;
- индексирование документов в некоем нетекстовом формате (например, при хранении PDF-документов внутри базы данных);
- и т.д.

Традиционно рассмотрим примерную структуру обсуждаемого индекса (рисунок 2.4.t), но ещё раз подчеркнём, что структура здесь совершенно классическая, а вся суть индекса — в том, как (с помощью вашего кода) вычисляются соответствующие индексируемые значения.

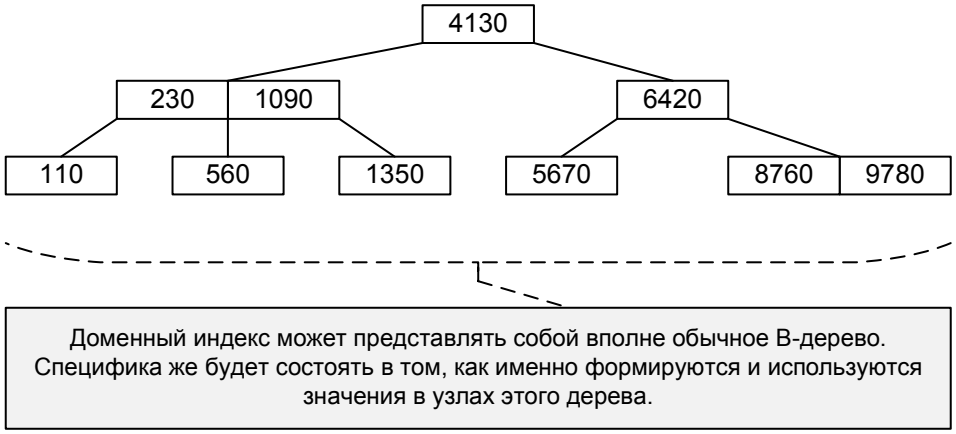


*Рисунок 2.4.t — **Схематичное представление доменного индекса***

---

[99] **Domain index** — an application-specific index that is used for indexing data in application-specific domains; a domain index is an instance of an index which is created, managed, and accessed by routines supplied by an indextype; this is in contrast to B-tree indexes maintained by Oracle internally («Building Domain Indexes»). [https://docs.oracle.com/cd/B10501_01/appdev.920/a96595/dci07idx.htm]

[100] Пример кода реализации доменного индекса (части `indextype`): https://docs.oracle.com/cd/B28359_01/server.111/b28286/ap_examples001.htm#SQLRF55430

На текущий момент доменные индексы в таком виде поддерживаются только в Oracle[101], но (учитывая тенденцию развития СУБД) есть основания ожидать в скором времени поддержки аналогичных решений и другими СУБД.

Прочитать много полезного про доменные (и другие) индексы в Oracle можно в книге «Expert Indexing in Oracle Database 11g: Maximum Performance for Your Database» (Bill Padfield, Sam R. Alapati, Darl Kuhn).

XML-индекс (XML index[102]) — индекс, оптимизированный для обработки XML-данных и ускорения поиска по путям и значениям внутри XML-документов.

Упрощённо: индекс для ускорения работы с XML.

Несмотря на то, что изначально не предполагалось с использованием реляционных СУБД хранить и обрабатывать иерархические структуры данных, в последние годы почти не осталось реляционных СУБД без поддержки XML и/или JSON[103]. Появление этих новых типов данных привело к необходимости их эффективного индексирования. Так появились XML-индексы.

Дальнейшие рассуждения здесь приведём на примере MS SQL Server. В данной СУБД индексирование XML построено на основе двух типов XML-индексов:

- Первичный XML-индекс представляет собой результат разбора хранимых XML-документов (что позволяет не производить такой разбор каждый раз при обращении к данным таблицы). Структурно такой индекс представляет собой ничто иное, как таблицу (т.н. `node table`) и предполагает хранение для каждого документа большого количества записей (примерно равного количеству элементов в XML-документе).
- Вторичные XML-индексы куда более похожи на «классические индексы» и позволяют ускорять поиск по указанным XPath и значениям элементов XML-документа.

Примерная структура того, как связаны между собой эти индексы (а также первичный кластерный индекс таблицы) показана на рисунке 2.4.u.

---

101 Похожие, но не идентичные возможности с недавних пор появились также в PostgreSQL. [https://www.postgresql.org]

102 **XML index** — index that can be created on xml data type columns (it indexes all tags, values and paths over the XML instances in the column and benefits query performance) («XML Indexes»). [https://msdn.microsoft.com/en-us/library/ms191497.aspx]

103 **JSON** — an open standard file format, and data interchange format, that uses human-readable text to store and transmit data objects consisting of attribute-value pairs and array data types (or any other serializable value). («Wikipedia») [https://en.wikipedia.org/wiki/JSON]

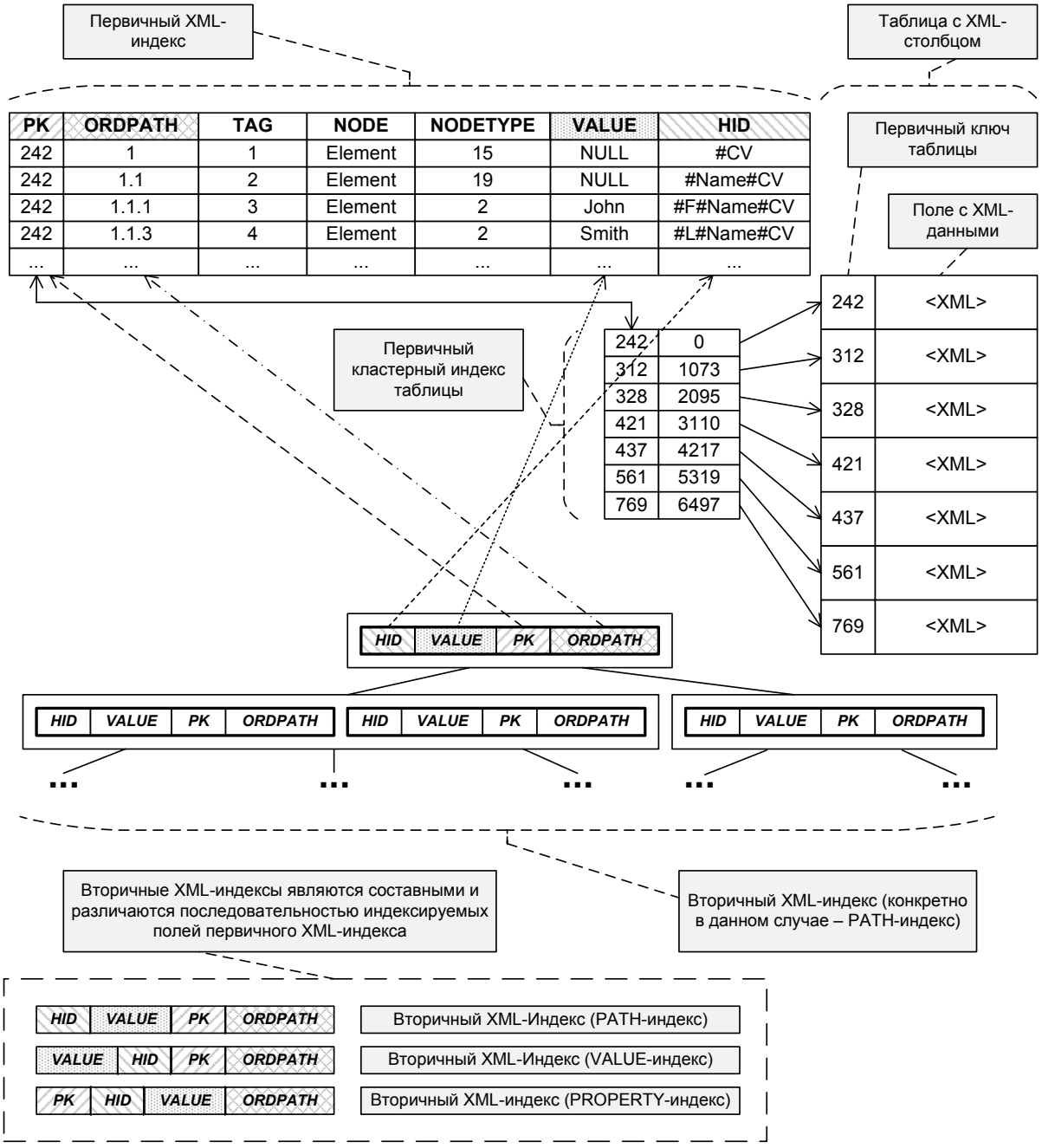


*Рисунок 2.4.u — **Схематичное представление XML-индекса***

XML-индексы явным образом реализованы в MS SQL Server (рисунок 2.4.u создан как раз для случая этой СУБД), но могут быть эмулированы в Oracle через индексы на значениях функций{129}. По слухам, в скором времени в MySQL также можно будет эмулировать XML-индексы через индексы на вычисляемых столбцах{129} (когда поддержка соответствующей технологии появится в данной СУБД).

Прочитать много полезного про XML-индексы в MS SQL Server можно в книге «Pro SQL Server 2008 XML» (Michael Coles).

Задание 2.4.a: какие индексы стоит построить на отношениях базы данных «Банк»{408}? Внесите соответствующие правки в модель.

Задание 2.4.b: стоит ли в базе данных «Банк»{408} использовать где бы то ни было составные индексы? Если вы считаете, что «да», внесите соответствующие правки в модель.

Задание 2.4.c: стоит ли в базе данных «Банк»{408} использовать кластерные индексы? Если вы считаете, что «да», внесите соответствующие правки в модель.

2.4.2. СОЗДАНИЕ И ИСПОЛЬЗОВАНИЕ ИНДЕКСОВ

С индексами на уровне базы данных и выполнения запросов к ней можно осуществить следующие основные операции:

- создать (удалить) индекс;
- отключить (включить) существующий индекс;
- проанализировать использование индекса при выполнении запроса;
- дать СУБД подсказку по использованию индекса.

Рассмотрим эти действия по порядку.

Создание и удаление индексов

Из материала предыдущей главы должно быть очевидно, что при наличии такого разнообразия индексов (в т.ч. специфичных для отдельных СУБД) нет единого универсального решения по их созданию. Иногда эту задачу даже приходится решать в несколько этапов, предварительно создавая дополнительные структуры базы данных и/или части индекса (например, это актуально для полнотекстовых{133} и XML-индексов{136}).

Потому здесь мы рассмотрим пример создания «просто индекса», который может ускорить поиск информации при выполнении тех или иных запросов.

Сначала создадим в трёх СУБД таблицу **books**[104] с первичным ключом **b_id**:

MySQL Создание таблицы books

```
CREATE TABLE `books`
(
 `b_id` INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
 `b_name` VARCHAR(150) NOT NULL,
 `b_year` SMALLINT UNSIGNED NOT NULL,
 `b_quantity` SMALLINT UNSIGNED NOT NULL,
 CONSTRAINT `PK_books` PRIMARY KEY (`b_id`)
)
```

MS SQL Создание таблицы books

```
CREATE TABLE [books]
(
 [b_id] [int] IDENTITY(1,1) NOT NULL,
 [b_name] [nvarchar](150) NOT NULL,
 [b_year] [smallint] NOT NULL,
 [b_quantity] [smallint] NOT NULL,
 CONSTRAINT [PK_books] PRIMARY KEY CLUSTERED ([b_id])
)
ON [PRIMARY]
```

Oracle Создание таблицы books

```
CREATE TABLE "books"
(
 "b_id"       NUMBER(10) NOT NULL,
 "b_name"     NVARCHAR2(150) NOT NULL,
 "b_year"     NUMBER(5) NOT NULL,
 "b_quantity" NUMBER(5) NOT NULL,
 PRIMARY KEY ("b_id")
)
ORGANIZATION INDEX
```

[104] См. базу данных «Библиотека» в книге «Работа с MySQL, MS SQL Server и Oracle в примерах» (С.С. Куликов) [http://svyatoslav.biz/database_book/]

Уже на данном этапе вы могли заметить, что во всех случаях первичный ключ является кластерным индексом{110} (в MySQL при использовании метода доступа{33} InnoDB первичный ключ автоматически становится кластерным индексом, в MS SQL Server мы явно создали первичный ключ как кластерный индекс, а в Oracle мы добились того же эффект через создание таблицы как индексно-организованной (опция `ORGANIZATION INDEX`)).

Но сейчас мы будем говорить не о преимуществах и недостатках использования кластерных индексов или индексной организации таблиц (увы, здесь очень сложно привести универсальные рецепты), а перейдём к «просто индексам».

Создадим три индекса, один из которых будет объективно избыточным:

- на полях `{b_name, b_year}` (составной индекс);
- на поле `b_quantity`;
- на поле `b_name` (этот индекс не имеет смысла, т.к. поле `b_name` является первым в составе индекса `{b_name, b_year}`, и ранее мы рассматривали{52}, что СУБД может производить поиск по первому полю в составном индексе столь же быстро, как и по всем полям в совокупности).

MySQL Создание индексов на таблице `books`

```
1   CREATE INDEX `idx_b_name_b_year`
2   ON `books` (`b_name`, `b_year`);
3
4   CREATE INDEX `idx_b_quantity`
5   ON `books` (`b_quantity`);
6
7   CREATE INDEX `idx_b_name`
8   ON `books` (`b_name`);
```

MS SQL Создание индексов на таблице `books`

```
1   CREATE INDEX [idx_b_name_b_year]
2   ON [books] ([b_name], [b_year]);
3
4   CREATE INDEX [idx_b_quantity]
5   ON [books] ([b_quantity]);
6
7   CREATE INDEX [idx_b_name]
8   ON [books] ([b_name]);
```

Oracle Создание индексов на таблице `books`

```
1   CREATE INDEX "idx_b_name_b_year"
2   ON "books" ("b_name", "b_year");
3
4   CREATE INDEX "idx_b_quantity"
5   ON "books" ("b_quantity");
6
7   CREATE INDEX "idx_b_name"
8   ON "books" ("b_name");
```

Из кода SQL-запросов очевидно, что в случае «просто индексов» (пока мы не начали управлять их видами и параметрами) во всех трёх СУБД синтаксис данной команды полностью идентичен.

Удаление индексов (в такой же простой ситуации, если мы не затрагиваем первичные ключи или сложноорганизованные индексы, опирающиеся на дополнительные структуры данных) столь же тривиально — достаточно выполнить команду `DROP INDEX`:

MySQL Удаление индекса

```
DROP INDEX `idx_b_name` ON `books`;
```

MS SQL Удаление индекса

```
DROP INDEX [idx_b_name] ON [books];
```

Oracle Удаление индекса

```
DROP INDEX "idx_b_name";
```

Обратите внимание: в Oracle имена индексов глобальны для всей схемы (именно поэтому не требуется указывать принадлежность индекса таблице), в то время как в MySQL и MS SQL Server имена индексов локальны в рамках таблицы (да, можно создать на разных таблицах индексы с одинаковыми именами, но делать этого не стоит, т.к. такой подход усложняет дальнейшую работы с базой данных).

Включение и отключение существующих индексов

По умолчанию индекс после создания сразу находится во включённом состоянии (enabled), однако в целях исследования производительности (или по иным причинам) может возникнуть потребность временно перевести индекс в отключённое состояние (disabled).

Выключение и повторное включение индекса имеет смысл именно с точки зрения исследования поведения СУБД (и то часто можно обойтись «подсказками{155}» по использованию индекса в SQL-запросе) — **не надо** отключать индексы перед объёмными операциями модификации данных: большинство СУБД обладает специальными механизмами выполнения таких операций без отключения индексов (см. далее{159} и документацию по конкретной СУБД).

Отключённый индекс автоматически становится устаревшим (информация в нём не соответствует реальному набору данных в таблице), и после включения требует перестроения (rebuild), т.е. достаточно трудоёмкой для СУБД операции.

Итак, если вы всё же решили отключить или включить индекс, это можно сделать следующими запросами.

Как следует из синтаксиса запроса для MySQL, отключить и включить отдельный индекс в этой СУБД нельзя. Также случае с MySQL команды **`DISABLE KEYS`** и **`ENABLE KEYS`** неприменимы к ныне самому популярному методу доступа{33} InnoDB: при их выполнении на таблице, использующей этот метод доступа, вы получите сообщение «Table storage engine for 'books' doesn't have this option». При использовании же метода доступа MyISAM, команды по отключению и включению индексов сработают (первичный ключ и уникальные индексы всё равно не будут отключены).

MySQL Отключение и включение индекса

```
-- Отключение индексов:
ALTER TABLE `books` DISABLE KEYS;

-- Включение индексов:
ALTER TABLE `books` ENABLE KEYS;
```

MS SQL Отключение и включение индекса

```
-- Отключение индекса:
ALTER INDEX [idx_b_name] ON [books] DISABLE;

-- Включение индекса:
ALTER INDEX [idx_b_name] ON [books] REBUILD;
```

Oracle Отключение и включение индекса

```
-- Без этой команды операции по модификации данных станут невозможными
-- для таблицы, на которой есть отключённые индексы:
ALTER SESSION SET skip_unusable_indexes = true;

-- Отключение индекса:
ALTER INDEX "idx_b_name" UNUSABLE;

-- Включение индекса:
ALTER INDEX "idx_b_name" REBUILD;
```

Несмотря на то, что MS SQL Server и Oracle предоставляют достаточно гибкие возможности по отключению и повторному включению отдельных индексов, на эти СУБД всё равно в полной мере распространяется ранее приведённое замечание: есть более технически эффективные способы получения желаемого поведения СУБД, потому считайте отключение индексов крайним средством.

Анализ использования индексов при выполнении запросов

Создавая индексы, мы предполагаем, что с их помощью СУБД сможет более эффективно выполнять те или иные запросы. Но всегда стоит проверить, оправдались ли наши надежды.

Одним из самых простых и в то же время рациональных способов такой проверки является анализ плана выполнения запроса, в котором СУБД предоставит нам информацию в т.ч. и об используемых индексах.

В различных СУБД способ получения плана выполнения запроса и содержимое такого плана сильно различаются, потому будем рассматривать несколько вариантов последовательно.

Для начала удалим все индексы (кроме первичных ключей) с ранее созданной таблицы **books** и добавим в эту таблицу десять миллионов записей. Теперь будем проверять, как добавление индексов повлияет на следующие запросы:

- подсчёт количества книг, изданных в указанном диапазоне лет;
- поиск книг с указанным названием;
- поиск книг с указанным годом издания и указанным словом в названии;
- поиск книг с максимальным количеством экземпляров;
- формирование списка из десяти книг с наименьшим количеством экземпляров.

В MySQL получить план запроса можно командой **EXPLAIN**, которую просто нужно добавить в начало запроса. Рассмотрим примеры запросов и предоставленных MySQL планов их выполнения.

MySQL Пример получения плана запроса

```
-- 1) Подсчёт количества книг, изданных в указанном диапазоне лет:
EXPLAIN
SELECT COUNT(*)
FROM   `books`
WHERE  `b_year` BETWEEN 1990 AND 1995;

-- 2) Поиск книг с указанным названием:
EXPLAIN
SELECT COUNT(*)
FROM   `books`
WHERE  `b_name` = 'Book 0 1';

-- 3) Поиск книг с указанным годом издания и указанным словом в названии:
EXPLAIN
SELECT COUNT(*)
FROM   `books`
WHERE  `b_year` = 1995
   AND `b_name` LIKE '%ok 0 5%';

-- 4) Поиск книг с максимальным количеством экземпляров:
EXPLAIN
SELECT *
FROM   `books`
WHERE  `b_quantity` = (SELECT MAX(`b_quantity`)
                       FROM   `books`);

-- 5) Формирование списка из десяти книг с наименьшим количеством экземпля-
ров:
EXPLAIN
SELECT *
FROM   `books`
ORDER  BY `b_quantity` ASC
LIMIT  10;
```

Для запросов 1-3 план выполнения оказывается одинаковым и выглядит следующим образом:

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	books	ALL	NULL	NULL	NULL	NULL	9730420	Using where

План выполнения запроса 4 таков:

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	books	ALL	NULL	NULL	NULL	NULL	9730420	Using where
2	SUBQUERY	books	ALL	NULL	NULL	NULL	NULL	9730420	NULL

План выполнения запроса 5 таков:

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	books	ALL	NULL	NULL	NULL	NULL	9730420	Using filesort

Из всей представленной здесь информации нас будут интересовать поля **possible_keys** (потенциально применимые индексы) и **key** (индекс, который был использован при выполнении запроса). Пока мы видим, что во всех случаях в этих полях находится значение **NULL**, что вполне логично, т.к. кроме первичных ключей никаких других индексов нет.

Получим планы выполнения запросов в MS SQL Server. В данной СУБД есть возможность просматривать план выполнения запроса как в текстовом виде[105], так и в графическом.

[105] См. подробности в документации: «Displaying Execution Plans by Using the Showplan SET Options (Transact-SQL)». [https://technet.microsoft.com/en-us/library/ms180765.aspx]

Поскольку графическое представление является более наглядным, текстовое приведём лишь для первого из следующих запросов.

MS SQL **Пример получения плана запроса**

```
-- Включение отображения плана выполнения запроса:
SET SHOWPLAN_TEXT ON

-- 1) Подсчёт количества книг, изданных в указанном диапазоне лет:
SELECT COUNT(*)
FROM   [books]
WHERE  [b_year] BETWEEN 1990 AND 1995;

-- 2) Поиск книг с указанным названием:
SELECT COUNT(*)
FROM   [books]
WHERE  [b_name] = 'Book 0 1';

-- 3) Поиск книг с указанным годом издания и указанным словом в названии:
SELECT COUNT(*)
FROM   [books]
WHERE  [b_year] = 1995
   AND [b_name] LIKE '%ok 0 5%';

-- 4) Поиск книг с максимальным количеством экземпляров:
SELECT *
FROM   [books]
WHERE  [b_quantity] = (SELECT MAX([b_quantity])
                       FROM   [books]);

-- 5) Формирование списка из десяти книг с наименьшим
-- количеством экземпляров:
SELECT TOP 10 *
FROM   [books]
ORDER  BY [b_quantity] ASC;
```

Итак, для первого запроса текстовое представление плана выполнения выглядит следующим образом:

```
Compute Scalar(DEFINE:([Expr1003]=CONVERT_IMPLICIT(int,[Expr1006],0)))
 |--Stream Aggregate(DEFINE:([Expr1006]=Count(*)))
  |--Clustered Index Scan(OBJECT:([book_theory].[dbo].[books].[PK_books]),
    WHERE:([book_theory].[dbo].[books].[b_year]>=[@1] AND
           [book_theory].[dbo].[books].[b_year]<=[@2]))
```

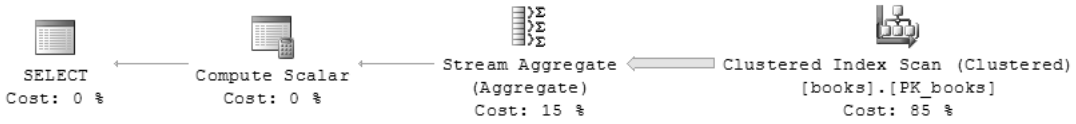
Если в MS SQL Server Management Studio выполнить запрос комбинацией клавиш Ctrl+L, мы не только получим красивое визуальное представление плана выполнения запроса (рисунок 2.4.v), но и дополнительную статистическую информацию по реально выполненным СУБД операциям, а также ещё и подсказки от СУБД относительно того, какие индексы могли бы ускорить выполнение запроса.

Вот пример такой подсказки по первому запросу:

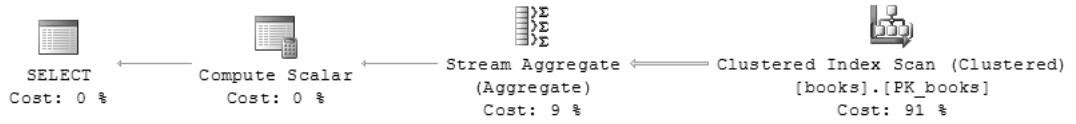
```
The Query Processor estimates that implementing the following index could improve the query
cost by 94.2711%.
CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [books] ([b_year])
```

Т.е. СУБД не только определяет нехватку индекса и потенциальную выгоду от его создания, но и формирует заготовку запроса для создания такого индекса.

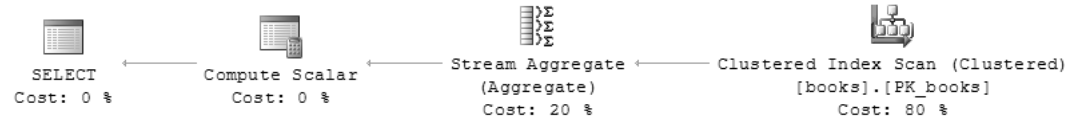
Запрос 1. Подсчёт количества книг, изданных в указанном диапазоне лет:



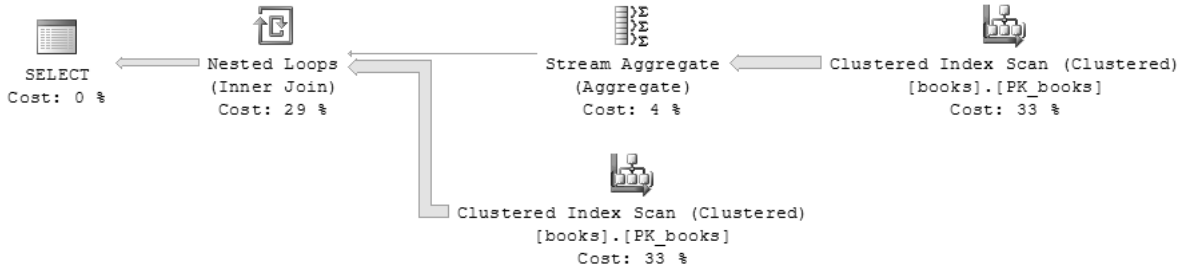
Запрос 2. Поиск книг с указанным названием:



Запрос 3. Поиск книг с указанным годом издания и указанным словом в названии:



Запрос 4. Поиск книг с максимальным количеством экземпляров:



Запрос 5. Формирование списка из десяти книг с наименьшим количеством экземпляров:

*Рисунок 2.4.v — **Визуальное представление планов выполнения запросов в MS SQL Server Management Studio***

Итак, во всех пяти планах мы видим только `Clustered Index Scan` (фактически — само чтение таблицы) и операции над полученными данными. И это вполне корректный результат, который демонстрирует, что пока СУБД нечего использовать для ускорения выполнения запроса.

В Oracle для получения плана выполнения запроса необходимо сначала добавить конструкцию `EXPLAIN PLAN FOR` в начало запроса, после чего воспользоваться одним из нескольких вариантов[106] просмотра полученного плана, например, таким:

Oracle **Один из вариантов просмотра полученного плана выполнения запроса**

```
SELECT PLAN_TABLE_OUTPUT
FROM TABLE(DBMS_XPLAN.DISPLAY())
```

Приведём планы выполнения следующих запросов:

[106] См. варианты просмотра полученного плана выполнения запроса здесь: https://docs.oracle.com/cd/E11882_01/server.112/e41573/ex_plan.htm#PFGRF94681

Oracle **Один из вариантов просмотра полученного плана выполнения запроса**

```
-- 1) Подсчёт количества книг, изданных в указанном диапазоне лет:
EXPLAIN PLAN FOR
SELECT COUNT(*)
FROM   "books"
WHERE  "b_year" BETWEEN 1990 AND 1995;

SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY());

-- 2) Поиск книг с указанным названием:
EXPLAIN PLAN FOR
SELECT COUNT(*)
FROM   "books"
WHERE  "b_name" = 'Book 0 1';

SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY());

-- 3) Поиск книг с указанным годом издания и указанным словом в названии:
EXPLAIN PLAN FOR
SELECT COUNT(*)
FROM   "books"
WHERE  "b_year" = 1995
   AND "b_name" LIKE '%ok 0 5%';

SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY());

-- 4) Поиск книг с максимальным количеством экземпляров:
EXPLAIN PLAN FOR
SELECT *
FROM   "books"
WHERE  "b_quantity" = (SELECT MAX("b_quantity")
                       FROM   "books");

SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY());

-- 5) Формирование списка из десяти книг с наименьшим
-- количеством экземпляров:
EXPLAIN PLAN FOR
SELECT *
FROM
 (
  SELECT "books".*,
         ROW_NUMBER() OVER (ORDER  BY "b_quantity" ASC) AS "rn"
  FROM   "books"
 ) "tmp"
WHERE "rn" <= 10;

SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY());
```

Полученные планы выполнения выглядят следующим образом (см. рисунок 2.4.w). Обратите внимание на тот факт, что эти «таблицы с результатами» на самом деле представляют собой просто текст, пусть и разбитый на отдельные строки колонки **PLAN_TABLE_OUTPUT**).

Запрос 1. Подсчёт количества книг, изданных в указанном диапазоне лет:

```
-------------------------------------------------------------------------------------------------
| Id  | Operation             | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |                     |     1 |     4 | 12197   (1)| 00:02:27 |
|   1 |  SORT AGGREGATE       |                     |     1 |     4 |            |          |
|*  2 |   INDEX FAST FULL SCAN| SYS_IOT_TOP_101259 |   492K|  1925K| 12197   (1)| 00:02:27 |
-------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
2 - filter(b_year>=1990 AND b_year<=1995)
```

Запрос 2. Поиск книг с указанным названием:

```
-------------------------------------------------------------------------------------------------
| Id  | Operation             | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |                     |     1 |    26 | 12193   (1)| 00:02:27 |
|   1 |  SORT AGGREGATE       |                     |     1 |    26 |            |          |
|*  2 |   INDEX FAST FULL SCAN| SYS_IOT_TOP_101259 |     2 |    52 | 12193   (1)| 00:02:27 |
-------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
2 - filter(b_name=U'Book 0 1')
```

Запрос 3. Поиск книг с указанным годом издания и указанным словом в названии:

```
-------------------------------------------------------------------------------------------------
| Id  | Operation             | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |                     |     1 |    30 | 12193   (1)| 00:02:27 |
|   1 |  SORT AGGREGATE       |                     |     1 |    30 |            |          |
|*  2 |   INDEX FAST FULL SCAN| SYS_IOT_TOP_101259 |  3499 |   102K| 12193   (1)| 00:02:27 |
-------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
2 - filter(b_year=1995 AND b_name LIKE U'%ok 0 5%')
```

Запрос 4. Поиск книг с максимальным количеством экземпляров:

```
-------------------------------------------------------------------------------------------------
| Id  | Operation              | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |                     | 40590 |  1545K| 24367   (1)| 00:04:53 |
|*  1 |  INDEX FAST FULL SCAN  | SYS_IOT_TOP_101259 | 40590 |  1545K| 12193   (1)| 00:02:27 |
|   2 |   SORT AGGREGATE       |                     |     1 |     4 |            |          |
|   3 |    INDEX FAST FULL SCAN| SYS_IOT_TOP_101259 |  8118K|    30M| 12174   (1)| 00:02:27 |
-------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
1 - filter(b_quantity= (SELECT MAX(b_quantity) FROM books books))
```

Запрос 5. Формирование списка из десяти книг с наименьшим количеством экземпляров:

```
-------------------------------------------------------------------------------------------------
| Id  | Operation               | Name               | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT        |                    |  8118K|  1579M|       | 93828   (1)| 00:18:46 |
|*  1 |  VIEW                   |                    |  8118K|  1579M|       | 93828   (1)| 00:18:46 |
|*  2 |   WINDOW SORT PUSHED RANK|                   |  8118K|   301M|   403M| 93828   (1)| 00:18:46 |
|   3 |    INDEX FAST FULL SCAN  | SYS_IOT_TOP_101259 |  8118K|   301M|       | 12174   (1)| 00:02:27 |
-------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
1 - filter(rn<=10)
2 - filter(ROW_NUMBER() OVER ( ORDER BY b_quantity)<=10)
```

*Рисунок 2.4.w — **Планы выполнения запросов в Oracle***

Как и в случае MySQL и MS SQL Server, у Oracle нет иного выхода, кроме сканирования таблицы целиком (**`INDEX FAST FULL SCAN`** на индексно-организованной таблице).

Пришло время создать индексы и проверить, как их наличие повлияет на планы выполнения запросов. Также для наглядности приведём сравнение медиан времени (после 100 итераций) выполнения запросов до и после создания индексов.

Запрос	Медиана времени выполнения до создания индексов, с			Медиана времени выполнения после создания индексов, с		
	MySQL	MS SQL	Oracle	MySQL	MS SQL	Oracle
Подсчёт количества книг, изданных в указанном диапазоне лет	6.228	0.963	3.082	0.074	0.012	0.177
Поиск книг с указанным названием	8.576	2.619	3.000	0.003	0.001	0.005
Поиск книг с указанным годом издания и указанным словом в названии	7.817	1.590	3.096	0.128	0.271	0.213
Поиск книг с максимальным количеством экземпляров	7.023	2.873	2.871	0.063	0.002	0.002
Формирование списка из 10 книг с наименьшим количеством экземпляров	10.464	6.450	5.653	0.003	0.001	5.573

Для проведения этого исследования были созданы индексы (синтаксис создания был рассмотрен ранее{139}):

- на поле `b_name`;
- на полях `{b_year, b_name}` (составной индекс, причём поля в нём следуют именно в таком порядке, т.к. уже существует отдельный индекс на поле `b_name`);
- на поле `b_quantity`.

Даже беглый взгляд на результаты эксперимента показывает, что использование индексов себя оправдало. Лишь в одном случае (пятый запрос в Oracle) результат «с индексами» почти не изменился. Стоит отметить, что на меньших объёмах данных (до миллиона записей) результаты «с индексами» в некоторых случаях могут стать даже хуже, чем «без индексов». Но к этому вопросу мы вернёмся чуть позже{154}.

Также подчеркнём ещё один интересный факт: в планах запросов от Oracle расчётное время выполнения запроса оказывается значительно большим, чем фактическое. И это считается корректным поведением оптимизатора запросов[107].

Переходим к сравнению планов выполнения запросов.

Для MySQL создание индексов привело к следующему изменению таких планов (см. код самих запросов выше{143}).

[107] См. обсуждение на AskTOM: https://asktom.oracle.com/pls/apex/f?p=100:11:0::::P11_QUESTION_ID:1434984500346471382

Запрос 1. Подсчёт количества книг, изданных в указанном диапазоне лет:

Было:

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	books	ALL	NULL	NULL	NULL	NULL	9730420	Using where

Стало:

id	select_type	table	type	possible_keys
1	SIMPLE	books	range	idx_b_year_b_name

key	key_len	ref	rows	Extra
idx_b_year_b_name	2	NULL	971120	Using where; Using index

Запрос 2. Поиск книг с указанным названием:

Было:

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	books	ALL	NULL	NULL	NULL	NULL	9730420	Using where

Стало:

id	select_type	table	type	possible_keys
1	SIMPLE	books	ref	idx_b_name

key	key_len	ref	rows	Extra
idx_b_name	452	const	5	Using where; Using index

Запрос 3. Поиск книг с указанным годом издания и указанным словом в названии:

Было:

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	books	ALL	NULL	NULL	NULL	NULL	9730420	Using where

Стало:

id	select_type	table	type	possible_keys
1	SIMPLE	books	ref	idx_b_year_b_name

key	key_len	ref	rows	Extra
idx_b_year_b_name	2	const	174654	Using where; Using index

Запрос 4. Поиск книг с максимальным количеством экземпляров:

Было:

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	books	ALL	NULL	NULL	NULL	NULL	9730420	Using where
2	SUBQUERY	books	ALL	NULL	NULL	NULL	NULL	9730420	NULL

Стало:

id	select_type	table	type	possible_keys
1	PRIMARY	books	ref	idx_b_quantity
2	SUBQUERY	NULL	NULL	NULL

key	key_len	ref	rows	Extra
idx_b_quantity	2	const	103390	Using where
NULL	NULL	NULL	NULL	Select tables optimized away

Запрос 5. Формирование списка из десяти книг с наименьшим количеством экземпляров:

Было:

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	books	ALL	NULL	NULL	NULL	NULL	9730420	Using filesort

Стало:

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	books	index	NULL	idx_b_quantity	2	NULL	10	NULL

Изменения в планах выполнения запросов позволяют сделать нам следующие выводы:

- Все созданные нами индексы сработали (т.е. СУБД сумела применить их или для поиска, или для извлечения данных).
- Количество данных, которые СУБД необходимо было обработать, сократилось весьма существенно (см. значения параметра **`rows`** — пусть оно и носит прогностический характер, изменения всё равно слишком существенны, чтобы их игнорировать).
- Ранее подробно рассмотренная мысль{52} о том, что СУБД может использовать отдельно первое поле в составном индексе для выполнения операции поиска столь же эффективно, как и весь индекс целиком, подтвердилась: в первом запросе составной индекс **`idx_b_year_b_name`** на полях **`{b_year, b_name}`** был использован MySQL для выполнения поиска по полю **`b_year`**.
- Наличие индекса позволяет СУБД очень эффективно оптимизировать даже запросы с подзапросами. Обратите внимание на значение параметра **Extra** в четвёртом запросе: значение «Select tables optimized away» в данном случае означает, что СУБД сумела получить всю необходимую информацию для выполнения подзапроса без обращения к данным таблицы.
- Индекс может оказаться полезным даже в случае, если он не построен на поле, по которому происходит поиск. Так в пятом запросе мы видим, что СУБД не нашла ни одного индекса, пригодного для оптимизации поиска (в параметре **`possible_keys`** находится значение **`NULL`**), но всё равно использовала индекс **`idx_b_quantity`**, т.к. по этому полю происходит упорядочивание данных. Такое же поведение СУБД (использование индекса, отсутствующего в списке «подходящих для выполнения запроса») характерно для ситуации т.н. «покрывающих индексов{124}».

Для MS SQL Server создание индексов привело к следующему изменению планов выполнения запросов (см. код самих запросов выше{144}) — см. рисунок 2.4.x.

В отличие от MySQL, здесь изменения поначалу не так бросаются в глаза. Кажется, что для запросов 1-3 всё осталось прежним. Но если обратить внимание на надписи возле пиктограмм операций, можно заметить, что вместо «Clustered Index Scan» (т.е. просмотра всех данных таблицы) MS SQL Server переключился на «Index Seek» (поиск с использованием индекса), что является значительно более быстрой операцией и позволяет во многих случаях сократить время выполнения запроса в десятки и сотни раз.

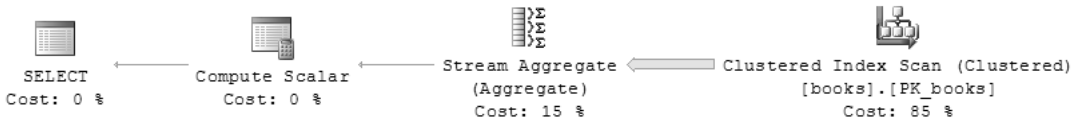
Для четвёртого запроса изменения ещё более существенны: вместо двух отдельных просмотров таблицы СУБД теперь определяет набор ключей искомых записей на основе индекса, затем на основе этих ключей выбирает сами записи (что тоже происходит намного быстрее, чем полный просмотр таблицы), и формирует итоговый результат (почти мгновенно, как следует из того, что на все три оставшиеся операции затрачено 0% общего времени выполнения запроса).

Наконец, в пятом запросе план выполнения изменился полностью: теперь вместо использования упорядочивания (объективно весьма долгой операции) СУБД идёт по тому же пути, что и в четвёртом запросе: на основе индекса получает перечень искомых записей, выясняет их ключи, по ключам ищет в таблице оставшиеся данные, после чего формирует итоговый результат.

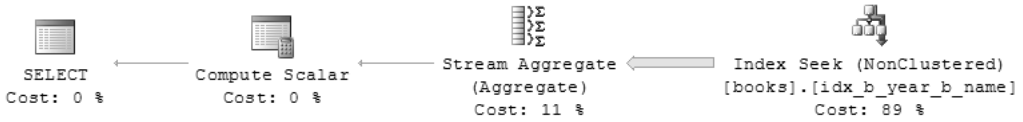
Выводы для данной СУБД в целом аналогичны выводам для MySQL: все индексы сработали, и в каждом случае СУБД смогла извлечь ощутимую пользу из их использования (даже в пятом запросе, где применение индекса могло показаться неочевидным, его наличие позволило весьма нетривиально перестроить план выполнения запроса и получить ощутимый прирост производительности).

Запрос 1. Подсчёт количества книг, изданных в указанном диапазоне лет:

Было:

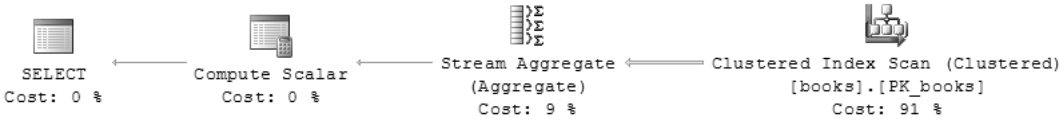


Стало:

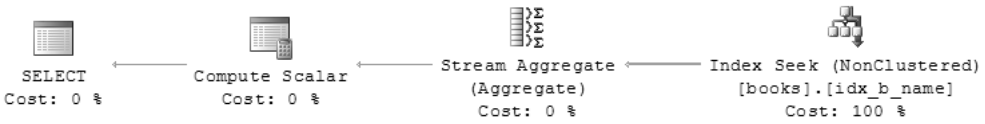


Запрос 2. Поиск книг с указанным названием:

Было:

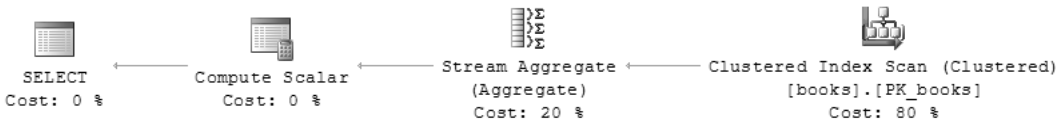


Стало:

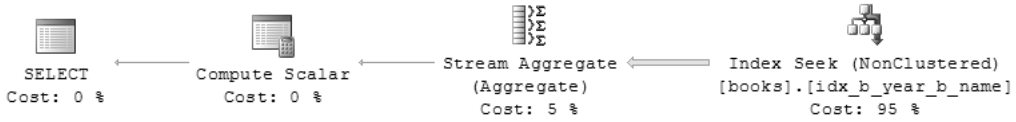


Запрос 3. Поиск книг с указанным годом издания и указанным словом в названии:

Было:

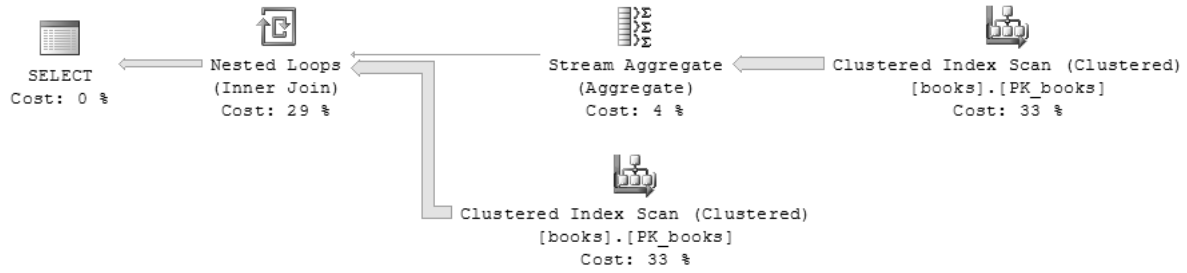


Стало:



Запрос 4. Поиск книг с максимальным количеством экземпляров:

Было:



Стало:

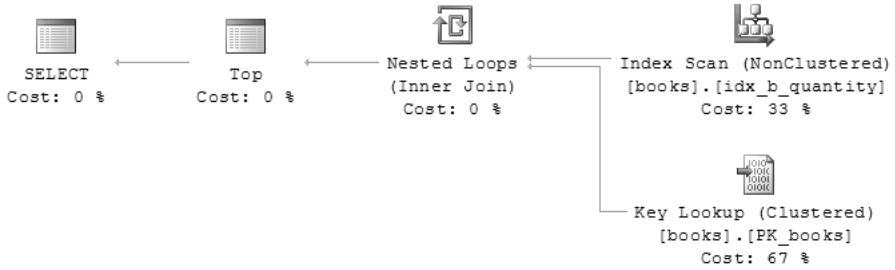


*Рисунок 2.4.x (часть 1) — **Визуальное представление планов выполнения запросов в MS SQL Server Management Studio до и после создания индексов***

Запрос 5. Формирование списка из десяти книг с наименьшим количеством экземпляров:
Было:

Стало:

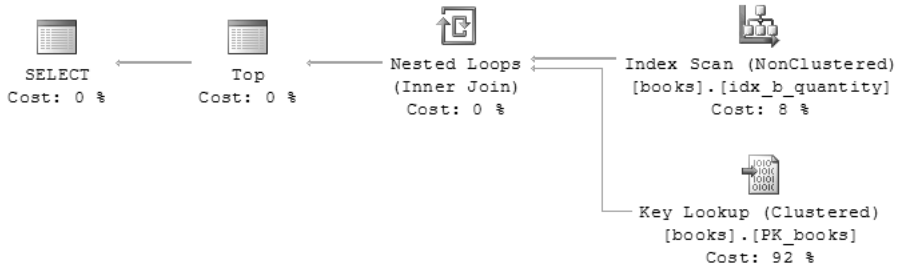


*Рисунок 2.4.x (часть 2) — **Визуальное представление планов выполнения запросов в MS SQL Server Management Studio до и после создания индексов***

Настало время подчеркнуть два важных факта:
1) Логика использования индексов очень сильно зависит от СУБД, метода доступа{33}, вида индекса, набора данных, запроса и иных специфичных факторов. Не ожидайте, что показанное в рассматриваемых примерах всегда и везде будет работать именно таким образом.
2) Помните, что работа с индексами не даётся СУБД «бесплатно»: они занимают оперативную память, снижают производительность операций модификации данных и приводят к иным накладным расходам. Создавайте индексы обдуманно — там, где они на самом деле нужны.

Переходим к сравнению планов выполнения запросов для Oracle. Для начала ещё раз внимательно посмотрите на результаты эксперимента по оценке производительности запросов{147}: Oracle — единственная СУБД, в которой медианное значение времени выполнения пятого запроса почти не изменилось.

Сравним представленные на рисунке 2.4.y планы выполнения запросов до и после создания индексов (см. код самих запросов выше{146}), чтобы понять, почему так происходит.

Из показанного на рисунке 2.4.y видно:

- Для запросов 1-4 вместо «INDEX FAST FULL SCAN» (что для индексно-организованных таблиц, фактически, означает чтение всех данных без учёта структуры индекса; здесь под словом «индекс» понимается сама индексно-организованная таблица **`books`**) СУБД применила «INDEX RANGE SCAN» (анализ индекса с учётом его структуры с целью получения идентификаторов записей; здесь под словом «индекс» понимается **`idx_b_year_b_name`** в первом запросе и **`idx_b_name`** во втором). Анализ индекса из одного-двух полей, да ещё и с учётом его (в нашем случае) древовидной структуры с последующей выборкой необходимых записей происходит быстрее, чем полное чтение данных таблицы.
- И лишь в случае пятого запроса мы видим, что его план не изменился. Т.е. в отличие от MySQL и MS SQL Server данная СУБД не смогла применить ни один из имеющихся индексов для оптимизации выполнения такого запроса. Основной причиной этого является принципиальное отличие механизмов выборки «первых N рядов» в Oracle.

Запрос 1. Подсчёт количества книг, изданных в указанном диапазоне лет:

Было:

```
-------------------------------------------------------------------------------------------------
| Id  | Operation             | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |                     |     1 |     4 | 12197   (1)| 00:02:27 |
|   1 |  SORT AGGREGATE       |                     |     1 |     4 |            |          |
|*  2 |   INDEX FAST FULL SCAN| SYS_IOT_TOP_101259  |   492K|  1925K| 12197   (1)| 00:02:27 |
-------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
2 - filter(b_year>=1990 AND b_year<=1995)
```

Стало:

```
-----------------------------------------------------------------------------------------
| Id  | Operation         | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |                   |     1 |     4 |  4072   (1)| 00:00:49 |
|   1 |  SORT AGGREGATE   |                   |     1 |     4 |            |          |
|*  2 |   INDEX RANGE SCAN| idx_b_year_b_name |   492K|  1925K|  4072   (1)| 00:00:49 |
-----------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
2 - access(b_year>=1990 AND b_year<=1995)
```

Запрос 2. Поиск книг с указанным названием:

Было:

```
-------------------------------------------------------------------------------------------------
| Id  | Operation             | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |                     |     1 |    26 | 12193   (1)| 00:02:27 |
|   1 |  SORT AGGREGATE       |                     |     1 |    26 |            |          |
|*  2 |   INDEX FAST FULL SCAN| SYS_IOT_TOP_101259  |     2 |    52 | 12193   (1)| 00:02:27 |
-------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
2 - filter(b_name=U'Book 0 1')
```

Стало:

```
-----------------------------------------------------------------------------
| Id  | Operation         | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |            |     1 |    26 |     4   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE   |            |     1 |    26 |            |          |
|*  2 |   INDEX RANGE SCAN| idx_b_name |     2 |    52 |     4   (0)| 00:00:01 |
-----------------------------------------------------------------------------
Predicate Information (identified by operation id):
2 - access(b_name=U'Book 0 1')
```

Запрос 3. Поиск книг с указанным годом издания и указанным словом в названии:

Было:

```
-------------------------------------------------------------------------------------------------
| Id  | Operation             | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |                     |     1 |    30 | 12193   (1)| 00:02:27 |
|   1 |  SORT AGGREGATE       |                     |     1 |    30 |            |          |
|*  2 |   INDEX FAST FULL SCAN| SYS_IOT_TOP_101259  |  3499 |   102K| 12193   (1)| 00:02:27 |
-------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
2 - filter(b_year=1995 AND b_name LIKE U'%ok 0 5%')
```

Стало:

```
-----------------------------------------------------------------------------------------
| Id  | Operation         | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |                   |     1 |    30 |   581   (1)| 00:00:07 |
|   1 |  SORT AGGREGATE   |                   |     1 |    30 |            |          |
|*  2 |   INDEX RANGE SCAN| idx_b_year_b_name |  3499 |   102K|   581   (1)| 00:00:07 |
-----------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
2 - access(b_year=1995)
filter(b_name LIKE U'%ok 0 5%')
```

*Рисунок 2.4.у (часть 1) — **Планы выполнения запросов в Oracle до и после создания индексов***

Запрос 4. Поиск книг с максимальным количеством экземпляров:

Было:

```
-------------------------------------------------------------------------------------------------
| Id  | Operation              | Name               | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |                    | 40590 |  1545K| 24367   (1)| 00:04:53 |
|*  1 |  INDEX FAST FULL SCAN  | SYS_IOT_TOP_101259 | 40590 |  1545K| 12193   (1)| 00:02:27 |
|   2 |   SORT AGGREGATE       |                    |     1 |     4 |            |          |
|   3 |    INDEX FAST FULL SCAN| SYS_IOT_TOP_101259 |  8118K|    30M| 12174   (1)| 00:02:27 |
-------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
1 - filter(b_quantity= (SELECT MAX(b_quantity) FROM books books))
```

Стало:

```
-------------------------------------------------------------------------------------------------
| Id  | Operation                   | Name               | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |                    | 40590 |  1545K| 12196   (1)| 00:02:27 |
|*  1 |  INDEX FAST FULL SCAN       | SYS_IOT_TOP_101259 | 40590 |  1545K| 12193   (1)| 00:02:27 |
|   2 |   SORT AGGREGATE            |                    |     1 |     4 |            |          |
|   3 |    INDEX FULL SCAN (MIN/MAX)| idx_b_quantity     |     1 |     4 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
1 - filter(b_quantity= (SELECT MAX(b_quantity) FROM books books))
```

Запрос 5. Формирование списка из десяти книг с наименьшим количеством экземпляров:

Было:

```
-------------------------------------------------------------------------------------------------------
| Id  | Operation                | Name               | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |                    |  8118K|  1579M|       | 93828   (1)| 00:18:46 |
|*  1 |  VIEW                    |                    |  8118K|  1579M|       | 93828   (1)| 00:18:46 |
|*  2 |   WINDOW SORT PUSHED RANK|                    |  8118K|   301M|   403M| 93828   (1)| 00:18:46 |
|   3 |    INDEX FAST FULL SCAN  | SYS_IOT_TOP_101259 |  8118K|   301M|       | 12174   (1)| 00:02:27 |
-------------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
1 - filter(rn<=10)
2 - filter(ROW_NUMBER() OVER ( ORDER BY b_quantity)<=10)
```

Стало:

```
-------------------------------------------------------------------------------------------------------
| Id  | Operation                | Name               | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |                    |  8118K|  1579M|       | 93828   (1)| 00:18:46 |
|*  1 |  VIEW                    |                    |  8118K|  1579M|       | 93828   (1)| 00:18:46 |
|*  2 |   WINDOW SORT PUSHED RANK|                    |  8118K|   301M|   403M| 93828   (1)| 00:18:46 |
|   3 |    INDEX FAST FULL SCAN  | SYS_IOT_TOP_101259 |  8118K|   301M|       | 12174   (1)| 00:02:27 |
-------------------------------------------------------------------------------------------------------
1 - filter(rn<=10)
2 - filter(ROW_NUMBER() OVER ( ORDER BY b_quantity)<=10)
```

*Рисунок 2.4.y (часть 2) — **Планы выполнения запросов в Oracle до и после создания индексов***

Вернёмся к вопросу{148} о том, почему на небольших объёмах данных создание индексов может оказывать гораздо меньший положительный (а иногда и вовсе отрицательный) эффект на скорость выполнения запросов.

Если в двух словах пересказать несколько сот страниц документации и заметок с AskTOM[108]: на малых объёмах данных слишком большое влияние на результаты эксперимента оказывают как внешние факторы (работа операционной системы, сторонних приложений, аппаратного обеспечения, сети и т.д.), так и внутренние (не отражаемые в плане запроса) особенности поведения СУБД.

Если многократно повторять этот эксперимент{147} на небольшом объёме данных, можно получить кратковременное увеличение медианного значения времени выполнения почти всех запросов. Однако, тенденция к его снижению явно просматривается по мере того, как данных в таблице становится всё больше и больше.

---

108 «Ask The Oracle Masters (AskTOM)» [https://asktom.oracle.com]

Отсюда следует ещё один крайне важный вывод: тестовые данные, на которых вы производите экспериментальную оценку производительности СУБД при выполнении тех или иных операций в тех или иных условиях, должны быть максимально приближены к реальным. Т.к. от набора данных зависит не только реалистичность результатов, но и даже само поведение СУБД (использованные ею алгоритмы и последовательность их применения), а потому решение, эффективное в одной ситуации, может оказаться губительным в другой.

Подсказки СУБД по использованию индексов

Применяйте подсказки по использованию индексов (index hints) равно как и подсказки по выполнению запросов (query hints) лишь в том случае, если вы совершенно точно уверены в том, что делаете (или для проведения исследований). Скорее всего, СУБД без вашей помощи построит более оптимальный план выполнения запроса.

Ранее{141} мы уже говорили о том, что отключение и повторное включение индексов в повседневной работе — плохая идея, вместо которой можно использовать специальные решения (см. далее{159}) и/или подсказки (hints), с помощью которых мы можем повлиять на использование СУБД индексов.

Традиционно начнём с MySQL. В данной СУБД нам доступно три варианта подсказок по использованию индексов. Мы можем:

- перечислить индексы, набором которых СУБД должна ограничиться при выполнении запроса (**`USE INDEX`**);
- перечислить индексы, которые запрещено использовать при выполнении запроса (**`IGNORE INDEX`**);
- указать индексы, которые обязательно нужно использовать как альтернативу недопустимо долгому просмотру таблицы (**`FORCE INDEX`**).

Также мы можем указать, для какого вида операции (**`JOIN`**, **`ORDER BY`**, **`GROUP BY`**) предполагается использовать индекс[109].

Продемонстрируем изменение плана запроса «Подсчёт количества книг, изданных в указанном диапазоне лет» (см. предыдущий параграф{142}) с применением подсказки **`IGNORE INDEX`**.

MySQL **Запрет использования индекса при выполнении запроса**

```
-- Подсчёт количества книг, изданных в указанном диапазоне лет
-- (без индекса):
EXPLAIN
SELECT COUNT(*)
FROM   `books`
IGNORE INDEX (`idx_b_year_b_name`)
WHERE  `b_year` BETWEEN 1990 AND 1995
```

До отказа от использования индекса план выполнения запроса выглядел таким образом:

id	select_type	table	type	possible_keys
1	SIMPLE	books	range	idx_b_year_b_name

Key	key_len	ref	rows	Extra
idx_b_year_b_name	2	NULL	971120	Using where; Using index

109 См. подробности в документации: «Index Hints» [https://dev.mysql.com/doc/refman/8.0/en/index-hints.html]

После отказа от использования индекса план выполнения запроса стал выглядеть таким образом (словно никаких доступных для выполнения данного запроса индексов и нет):

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	books	ALL	NULL	NULL	NULL	NULL	9730420	Using where

Таким простым изменением запроса в MySQL можно быстро проверить, по-прежнему ли наличие индекса позволяет ускорить выполнение запроса, или же ситуация изменилась настолько, что индекс более неэффективен и может быть удалён.

Если же мы попытаемся заставить (**`FORCE INDEX`**) СУБД использовать неподходящий для выполнения запроса индекс, ситуация будет аналогичной.

MySQL **Принудительное использование индекса при выполнении запроса**

```
-- Подсчёт количества книг, изданных в указанном диапазоне лет
-- (с принудительным использованием неподходящего индекса):
EXPLAIN
SELECT COUNT(*)
FROM   `books`
FORCE  INDEX (`idx_b_quantity`)
WHERE  `b_year` BETWEEN 1990 AND 1995
```

До принудительного использования неподходящего индекса план выполнения запроса выглядел таким образом:

id	select_type	table	type	possible_keys
1	SIMPLE	books	range	idx_b_year_b_name

Key	key_len	ref	rows	Extra
idx_b_year_b_name	2	NULL	971120	Using where; Using index

После принудительного использования неподходящего индекса план выполнения запроса стал выглядеть таким образом:

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	books	ALL	NULL	NULL	NULL	NULL	9730420	Using where

В данном случае, несмотря на все наши усилия, СУБД всё же не поддалась на провокацию и использовала единственно доступный способ выполнения запроса — просмотр всей таблицы. Но в более сложных ситуациях использование данной подсказки может привести к формированию ещё более неоптимального плана.

Эта картина лишь подтверждает приведённое в начале данного параграфа предостережение: неверными подсказками мы можем навредить работе СУБД, что может быть даже хорошо в исследовательских задачах, но недопустимо в работе реальных приложений.

В MS SQL Server подсказки по использованию индексов не являются обособленной частью синтаксиса и размыты по общей структуре подсказок по выполнению запросов[110].

В качестве примера реализуем с использованием данной СУБД поведение аналогичное поведению MySQL.

Запретим MS SQL Server использовать индекс при выполнении запроса «Подсчёт количества книг, изданных в указанном диапазоне лет» (см. предыдущий параграф{142}).

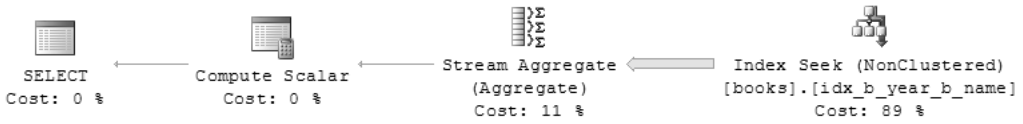
[110] См. подробности в документации: «Hints (Transact-SQL)» [https://docs.microsoft.com/en-us/sql/t-sql/queries/hints-transact-sql]

MS SQL Запрет использования индекса при выполнении запроса

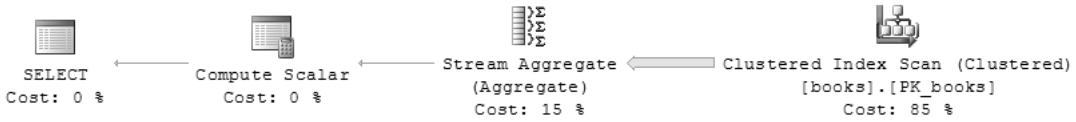
```
-- Подсчёт количества книг, изданных в указанном диапазоне лет
-- (без индекса):
SELECT COUNT(*)
FROM   [books]
WITH   (INDEX(0))
WHERE  [b_year] BETWEEN 1990 AND 1995
```

Как и в случае с MySQL, здесь ситуации до отказа от использования индекса и после отказа представляют собой точную копию ситуаций «индекс есть» и «индекса нет».

План выполнения запроса до отказа от использования индекса:



План выполнения запроса после отказа от использования индекса:

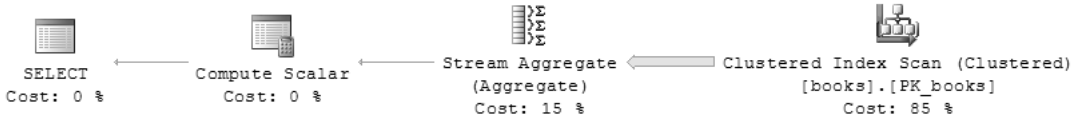


Попытаемся заставить СУБД использовать индекс, бесполезный при выполнении указанного запроса.

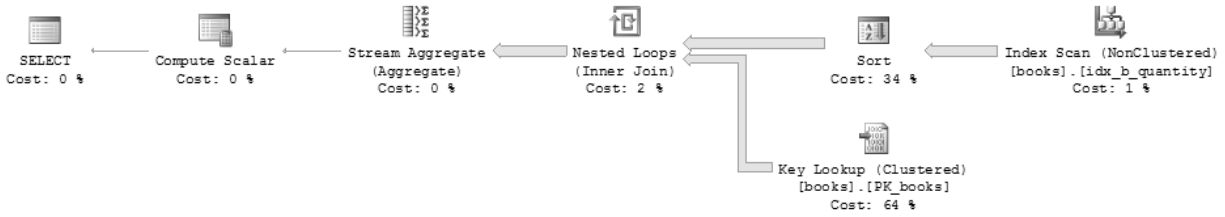
MS SQL Принудительное использование индекса при выполнении запроса

```
-- Подсчёт количества книг, изданных в указанном диапазоне лет
-- (с принудительным использованием неподходящего индекса):
EXPLAIN
SELECT COUNT(*)
FROM   `books`
FORCE  INDEX (`idx_b_quantity`)
WHERE  `b_year` BETWEEN 1990 AND 1995
```

До принудительного использования неподходящего индекса план выполнения запроса выглядел таким образом:



После принудительного использования неподходящего индекса план выполнения запроса стал выглядеть таким образом:



Здесь (в отличие от MySQL) мы как раз получили ситуацию заметного ухудшения плана выполнения запроса, когда MS SQL Server постарался выполнить нашу подсказку, но в итоге лишь совершил несколько бессмысленных лишних операций.

В Oracle ситуация с подсказками по использованию индексов полностью аналогична ситуации с MS SQL Server в том плане, что эти подсказки являются частями общей структуры подсказок оптимизатору выполнения запросов[111].

В качестве примера реализуем с использованием данной СУБД поведение аналогичное поведению MySQL и MS SQL Server.

Запретим Oracle использовать индекс при выполнении запроса «Подсчёт количества книг, изданных в указанном диапазоне лет» (см. предыдущий параграф{142}).

Oracle **Запрет использования индекса при выполнении запроса**

```
1   -- Подсчёт количества книг, изданных в указанном диапазоне лет
2   -- (без индекса):
3   EXPLAIN PLAN FOR
4   SELECT /*+ NO_INDEX("books" "idx_b_year_b_name") */
5          COUNT(*)
6   FROM   "books"
7   WHERE  "b_year" BETWEEN 1990 AND 1995;
8
9   SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY());
```

И вот уже в третий раз мы наблюдаем картину, в которой планы выполнения запросов «с индексом» и «без индекса» меняются местами.

До отказа от использования индекса план выполнения запроса выглядел таким образом:

```
-------------------------------------------------------------------------------------------
| Id  | Operation         | Name               | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |                    |     1 |     4 |  4072   (1)| 00:00:49 |
|   1 |  SORT AGGREGATE   |                    |     1 |     4 |            |          |
|*  2 |   INDEX RANGE SCAN| idx_b_year_b_name  |   492K|  1925K|  4072   (1)| 00:00:49 |
-------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
2 - access(b_year>=1990 AND b_year<=1995)
```

После отказа от использования индекса план выполнения запроса стал выглядеть таким образом (словно никаких доступных для выполнения данного запроса индексов и нет):

```
-------------------------------------------------------------------------------------------
| Id  | Operation             | Name               | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |                    |     1 |     4 | 12197   (1)| 00:02:27 |
|   1 |  SORT AGGREGATE       |                    |     1 |     4 |            |          |
|*  2 |   INDEX FAST FULL SCAN| SYS_IOT_TOP_101259 |   492K|  1925K| 12197   (1)| 00:02:27 |
-------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
2 - filter(b_year>=1990 AND b_year<=1995)
```

Теперь попытаемся заставить СУБД использовать индекс, бесполезный при выполнении текущего запроса.

Oracle **Принудительное использование индекса при выполнении запроса**

```
1   -- Подсчёт количества книг, изданных в указанном диапазоне лет
2   -- (с принудительным использованием неподходящего индекса):
3   EXPLAIN PLAN FOR
4   SELECT /*+ INDEX("books" "idx_b_quantity") */
5          COUNT(*)
6   FROM   "books"
7   WHERE  "b_year" BETWEEN 1990 AND 1995;
8
9   SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY());
```

[111] См. подробности в документации: «Using Optimizer Hints»: https://docs.oracle.com/cd/E11882_01/server.112/e41573/hintsref.htm.

До принудительного использования неподходящего индекса план выполнения запроса выглядел таким образом:

```
-----------------------------------------------------------------------------------------
| Id  | Operation         | Name               | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |                    |     1 |     4 |  4072   (1)| 00:00:49 |
|   1 |  SORT AGGREGATE   |                    |     1 |     4 |            |          |
|*  2 |   INDEX RANGE SCAN| idx_b_year_b_name  |   492K|  1925K|  4072   (1)| 00:00:49 |
-----------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
2 - access(b_year>=1990 AND b_year<=1995)
```

После принудительного использования неподходящего индекса план выполнения запроса стал выглядеть таким образом:

```
-----------------------------------------------------------------------------------------
| Id  | Operation          | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |                     |     1 |     4 |  6748K  (1)| 22:29:43 |
|   1 |  SORT AGGREGATE    |                     |     1 |     4 |            |          |
|*  2 |   INDEX UNIQUE SCAN| SYS_IOT_TOP_101259  |   492K|  1925K|  6748K  (1)| 22:29:43 |
|   3 |    INDEX FULL SCAN | idx_b_quantity      |  8118K|       | 29911   (1)| 00:05:59 |
-----------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
2 - filter(b_year>=1990 AND b_year<=1995)
```

Как и в случае с MS SQL Server, здесь мы снова получили ситуацию заметного ухудшения плана выполнения запроса, когда СУБД совершила несколько бессмысленных лишних операций. Обратите внимание на прогностические показатели времени выполнения операций верхнего уровня: было 49 секунд, стало 22 с половиной часа. Стоит задуматься.

После рассмотрения данных примеров может сложиться мнение, что подсказками СУБД по использованию индексов можно только навредить (в общем — это правильное мнение, которое убережёт вас от множества проблем).

На самом деле, в действительно сложных ситуациях при наличии достаточного опыта и глубокого понимания структуры базы данных, особенностей конкретного запроса и ещё множества фактов применение подсказок СУБД по использованию индексов может значительно улучшить план выполнения запроса. Но столь сложные и при этом реалистичные задачи выходят далеко за рамки данной книги.

И в завершение этого раздела поясним, как бороться с проблемой снижения производительности модификации данных при наличии большого количества индексов. Чаще всего этот вопрос возникает в контексте операции вставки, хоть многие решения будут полезны и при выполнении операций обновления и удаления данных.

Для каждого из приведённых ниже советов актуально предостережение: в контексте выполняемой вами задачи тот или иной подход может оказаться неприменимым, технически невозможным или несовместимым с используемой вами СУБД, методом доступа{33}, схемой базы данных и т.д.

И всё же.

- Удостоверьтесь, что на вашей таблице нет дублирующихся индексов (помните{52}, что по первому полю составного индекса СУБД умеет производить поиск столь же быстро, как и по всему составному индексу — потому нет необходимости создавать отдельный индекс на этом поле).
- Выполняйте несколько операций модификации в рамках одной транзакции{374}. В большинстве случаев СУБД не будет обновлять индексы до завершения транзакции.

- Убедитесь, что вы оптимально выбрали кластерный индекс{110} модифицируемой таблицы и выполняйте операции модификации так, чтобы значения соответствующего поля в модифицируемых данных шли в той же последовательности, что и значения этого же поля в кластерном индексе.
- Рассмотрите возможность использования временных таблиц как промежуточного буфера для хранения обрабатываемой информации, максимально перенесите на сторону СУБД любую сопутствующую операции модификации обработку данных.
- Если модифицируемая таблица содержит внешние ключи{47}, убедитесь, что соответствующие поля (или совокупности полей) родительских таблиц проиндексированы (хоть это условие почти всегда и выполняется автоматически).
- Если ваша СУБД позволяет, временно отключите проверку значений внешних ключей{47} и уникальных индексов{109}. Это **очень** опасная операция, и потому трижды подумайте, прежде чем выполнять её.
- Изучите план выполнения запроса и соответствующие рекомендации по вашей СУБД. Как правило, там есть много интересного[112, 113, 114].

На этом раздел, посвящённый основным структурам реляционных баз данных завершён. В следующем разделе мы, опираясь на полученные знания, рассмотрим способы построения моделей баз данных, устойчивых к ряду неочевидных ошибок при выполнении операций чтения и модификации данных.

Задание 2.4.d: какие индексы в базе данных «Банк»{408} следует отключить при импорте большого объёма данных? Насколько это повысит производительность? Проведите соответствующий эксперимент.

Задание 2.4.e: изучите несколько планов выполнения типичных запросов к базе данных «Банк»{408}. Можно ли ускорить выполнение этих запросов использованием «подсказок СУБД по использованию индексов{155}»? Проведите соответствующий эксперимент.

Задание 2.4.f: можно ли при создании базы данных «Банк»{412} использовать альтернативный код для формирования индексов? Если вы считаете, что «да», напишите соответствующее решение и проверьте его работоспособность.

112 Для MySQL: https://dev.mysql.com/doc/refman/8.0/en/optimizing-innodb-bulk-data-loading.html

113 Для MS SQL Server: https://technet.microsoft.com/en-us/library/ms190421(v=sql.105).aspx

114 Для Oracle: http://www.dba-oracle.com/t_insert_tuning.htm

НОРМАЛИЗАЦИЯ И НОРМАЛЬНЫЕ ФОРМЫ

АНОМАЛИИ ОПЕРАЦИЙ С ДАННЫМИ

3.1.1. ТЕОРЕТИЧЕСКИЙ ОБЗОР АНОМАЛИЙ ОПЕРАЦИЙ С ДАННЫМИ

Исторически ситуация с формулировкой строгого универсального определения аномалий операций с данными сложилась не лучшим образом: различные авторы чётко описывают отдельные виды аномалий, для общего определения оставляя лишь обтекаемые фразы вида «некоторые сложности[115]», «дополнительные проблемы[116]» и т.д. Потому обобщим описание отдельных аномалий (будет рассмотрено далее) и сформулируем общее определение.

Аномалии операций с данными (data operation anomalies) — некорректное выполнение операций с данными или возникновение побочных эффектов операций с данными, ставшее результатом нарушения требования адекватности базы данных предметной области{12}.

Упрощённо: выполняя правильную (с точки зрения пользователя) операцию правильным SQL-запросом, мы получаем ошибочный результат или нежелательный побочный эффект.

115 «… variety of… difficulties…» (C.J. Date, "An Introduction to Database Systems", 8th edition, p. 359).

116 «… additional problem… » (Ramez Elmasri, Shamkant Navathe, "Fundamentals of Database Systems", 6th edition, p. 507).

Традиционно выделяют следующие три вида аномалий операций с данными.

Аномалия вставки (insertion anomaly[117]) — нарушение логики вставки данных, выражающееся в необходимости использовать выдуманные значения или `NULL`-значения для части атрибутов.

Упрощённо: мы не знаем значений некоторых полей записи, и вынуждены или выдумывать их, или заменять `NULL`-значениями.

Аномалия удаления (deletion anomaly[118]) — нарушение логики удаления данных, выражающееся в потере не предназначенной для удаления информации, последняя копия которой хранилась в атрибутах удаляемой записи.

Упрощённо: мы хотели удалить одни данные, а вместе с ними потеряли и другие, удалять которые не собирались.

Аномалия обновления (modification anomaly[119]) — нарушение логики обновления данных, выражающееся в необходимости в целях сохранения консистентности базы данных обновлять значения некоторого атрибута в нескольких записях помимо той, с которой сейчас выполняется операция обновления.

Упрощённо: обновляя одну запись, мы вынуждены обновить и ещё несколько других, чтобы не нарушить целостность базы данных.

Также отметим, что пусть в научной литературе такого термина нет, мы можем говорить и об аномалии выборки, т.к. она очень часто сопутствует аномалии модификации.

Аномалия выборки (select anomaly) — нарушение логики выборки данных, выражающееся в искажении результатов выборки.

Упрощённо: мы выбираем какие-то данные, но или получаем их не в полном объёме, или вовсе получаем не те данные.

Важно понимать, что аномалии операций с данными ничего общего не имеют со сбоями в работе СУБД или ошибками в SQL-запросе. При возникновении аномалии СУБД работает совершенно корректно, и выполняемый SQL-запрос тоже корректен и соответствует поставленной цели.

В этом и кроется опасность аномалий: всё работает верно, а результаты получаются неправильными.

Продемонстрируем каждую из рассмотренных аномалий на примере, для которого используем представленное на рисунке 3.1.а отношение `work`.

---

117 **Insertion anomalies** can be differentiated into two types: a) To insert a new tuple into table, we must include either the attribute values for all columns or NULLs (if there is no data for some columns); b) To insert a new tuple into table, we must include NULL value into PK/FK fields, and that violates the entity integrity. (Ramez Elmasri, Shamkant Navathe, "Fundamentals of Database Systems", 6th edition, [with some text rephrasing])

118 **Deletion anomalies** are related to the next situation: if we delete from a tuple that happens to represent the last copy for a particular attribute value, the information concerning that value is lost from the database. (Ramez Elmasri, Shamkant Navathe, "Fundamentals of Database Systems", 6th edition, [with some text rephrasing])

119 **Modification anomalies** are related to the next situation: if we change the value of one of the attributes of a particular tuple, we must update the tuples of all corresponding tuples; otherwise, the database will become inconsistent. (Ramez Elmasri, Shamkant Navathe, "Fundamentals of Database Systems", 6th edition, [with some text rephrasing])

work

PK			
w_employee	**w_project**	**w_role**	**w_salary**
Иванов И.И.	Проект «Дуб»	Руководитель	500
Петров П.П.	Проект «Ясень»	Консультант	150
Сидоров С.С.	Проект «Клён»	Руководитель	900
Сидоров С.С.	Проект «Дуб»	Исполнитель	300

*Рисунок 3.1.a — **Отношение `work`, используемое для демонстрации аномалий операций с данными***

Аномалия вставки при работе с таким отношением может проявляться в следующих формах.

При принятии на работу нового сотрудника (см. рисунок 3.1.b, случай 1) мы обязаны сразу же указать, на каком проекте, в какой роли и с какой зарплатой он работает. Если хотя бы части этой информации у нас нет, её придётся выдумать или заменить на `NULL`. Но поскольку поле `w_project` является частью первичного ключа, как минимум в это поле вставка `NULL`-значения запрещена.

При появлении нового проекта (см. рисунок 3.1.b, случай 2) мы обязаны сразу же приписать к нему хотя бы одно сотрудника (и указать его зарплату).

При появлении новой роли (см. рисунок 3.1.b, случай 3) мы обязаны сразу же приписать к ней проект и сотрудника с зарплатой.

Таким образом, мы постоянно вынуждены использовать при вставке некую информацию, которой объективно может у нас не быть на момент выполнения операции. В редких случаях такая информация может содержаться в уже имеющихся в таблице записях, но в такой ситуации необходимо предварительно прочитать эту информацию, используя заранее реализованный (и, как правило, нетривиальный и ненадёжный) алгоритм.

work

PK				
w_employee	**w_project**	**w_role**	**w_salary**	
Иванов И.И.	Проект «Дуб»	Руководитель	500	
Петров П.П.	Проект «Ясень»	Консультант	150	
Сидоров С.С.	Проект «Клён»	Руководитель	900	
Сидоров С.С.	Проект «Дуб»	Исполнитель	300	
John Smith	**?**	**?**	**?**	1
?	Проект «Липа»	**?**	**?**	2
?	**?**	Контролёр	**?**	3

*Рисунок 3.1.b — **Пример аномалий вставки***

Аномалия удаления при работе с таким отношением может проявляться в следующих формах.

Если сотрудник «Петров П.П.» уволится (см. рисунок 3.1.c, случай 1), и соответствующая запись будет удалена, будет утеряна информация о том, что в фирме был проект «Ясень» и существовала роль «Консультант» (нигде, кроме этой записи данная информация не фигурирует).

Если в фирме упразднят роль «Руководитель» (см. рисунок 3.1.c, случай 2) и удалят соответствующие записи, будет утеряна информация о сотруднике «Иванов И.И.» (а если бы он был единственным, кто работает на проекте «Дуб», информация об этом проекте также была бы утеряна).

Если будет закрыт проект «Ясень» (см. рисунок 3.1.c, случай 3) и удалят соответствующие записи, будет утеряна информация о сотруднике «Петров П.П.» и роли «Консультант».

work

PK				
w_employee	**w_project**	**w_role**	**w_salary**	
Иванов И.И.	Проект «Дуб»	Руководитель	500	
Петров П.П.	Проект «Ясень»	Консультант	150	
Сидоров С.С.	Проект «Клён»	Руководитель	900	
Сидоров С.С.	Проект «Дуб»	Исполнитель	300	
John Smith	**?**	**?**	**?**	**1**
?	Проект «Липа»	**?**	**?**	**2**
?	**?**	Контролёр	**?**	**3**

*Рисунок 3.1.c — **Пример аномалий удаления***

Таким образом, удаляя некую одну часть данных, мы неожиданно для себя теряем другую часть данных, которую не собирались удалять.

Аномалия обновления при работе с таким отношением может проявляться в следующих формах.

Если будет решено переименовать роль «Руководитель» в «Координатор» (см. рисунок 3.1.d, случай 1), эту информацию придётся обновить в нескольких записях.

Если сотрудник «Сидоров С.С.» решит сменить ФИО на «Sidorov S.S.» (см. рисунок 3.1.d, случай 2), нужно будет неким образом понять, в каких записях речь идёт об одном и том же «Сидорове С.С.» (что вовсе не очевидно в такой структуре базы данных) и внести правку в эти записи. При этом важно гарантированно отличить этого переименовавшегося «Сидорова С.С.» от его полных тёзок, чтобы случайно не переименовать и их.

*Рисунок 3.1.d — **Пример аномалий обновления***

Таким образом, обновляя данные в одной записи, мы вынуждены обновлять эти же данные и в других записях, при этом рискуя случайно обновить то, что обновлять не стоило, или пропустить что-то, что обновить стоило.

Аномалия выборки, как правило, является следствием возникновения аномалии обновления.

Представим, что в процессе только что рассмотренного изменения ФИО сотрудника с «Сидоров С.С.» на «Sidorov S.S.» мы не обновили все необходимые записи — именно эта ситуация показана на рисунке 3.1.d (случай 2). Теперь, например, получая список проектов и суммарную зарплату этого сотрудника, мы получим результат, в котором будет отсутствовать информация о проекте «Дуб», а суммарная зарплата будет на 300 меньше, чем есть в реальности.

С аномалиями обновления и выборки связан ещё один печальный факт: их наличие свидетельствует о том, что (скорее всего) в соответствующем программном средстве, использующем такую базу данных, необходимые значения не выбираются из предложенного списка (проектов, сотрудников и т.д.), а вводятся каждый раз заново вручную. Такой подход резко повышает вероятность опечаток, т.е. введённое значение не будет совпадать с уже имеющимися, а потому в дальнейшем не попадёт в выборку и не будет обновлено или удалено при выполнении соответствующих операций.

Подведём краткий итог: если мы в реальной жизни попытаемся использовать только что рассмотренное отношение `work`, корректно не будет работать ничего — ни вставка, ни удаление, ни обновление, ни даже выборка данных.

Очевидно, необходимо внести некие изменения в схему данных, и именно об этом речь пойдёт дальше, когда мы будем говорить о нормализации.

А пока мы кратко рассмотрим, как в реальной жизни понять, что имеющаяся схема данных подвержена тем или иным аномалиям операций с данными.

Задание 3.1.a: какие аномалии вставки данных возможны в базе данных «Банк»{408}? Доработайте модель так, чтобы исключить возможность их возникновения.

Задание 3.1.b: какие аномалии обновления данных возможны в базе данных «Банк»{408}? Доработайте модель так, чтобы исключить возможность их возникновения.

Задание 3.1.c: какие аномалии обновления данных возможны в базе данных «Банк»{408}? Доработайте модель так, чтобы исключить возможность их возникновения.

3.1.2. СПОСОБЫ ВЫЯВЛЕНИЯ АНОМАЛИЙ ОПЕРАЦИЙ С ДАННЫМИ

Ярчайшим признаком того, что при работе с той или иной схемой базы данных будут возникать аномалии операций с данными, является нарушение нормальных форм. Но так мы рискуем получить замкнутый круг (т.к. одним из признаков нарушения нормальных форм является наличие аномалий операций с данными). Потому в данной главе мы рассмотрим иные, не затрагивающие нормализацию, способы выявления аномалий и признаки их наличия.

Признаки наличия аномалий операций с данными лучше всего выявляются в процессе проектирования базы данных.

При нисходящем проектировании{11} стоит постоянно проверять, не получилась ли такая ситуация, при которой в одно отношение попадает информация о нескольких независимых сущностях реального мира. Эта задача немного усложняется тем, что одни сущности реального мира могут быть свойствами других сущностей. Такое описание может звучать запутанно, потому поясним его на примерах.

Для начала рассмотрим случай, когда вывод очевиден. Допустим, мы создаём базу данных отдела кадров, описываем отношение **`employee`** и видим такую картину.

```
Employee:
  • First name
  • Middle name
  • Last name
  • Child name
```

Здесь совершенно объективно информация о ребёнке (сразу возникает вопрос, почему о *ребёнке*, а не о *детях*) напрасно размещена в атрибутах сущности **`employee`**. Для такой схемы будут характерны как минимум аномалии удаления{162} и обновления{162}.

Если же переработать эту схему следующим образом, данная проблема более не будет актуальна (в процессе дальнейшего проектирования базы данных между отношениями **`employee`** и **`child`** нужно будет установить связь «многие ко многим{60}»).

```
Employee:
  • First name
  • Middle name
  • Last name

Child:
  • Child name
```

Теперь рассмотрим случай, когда вывод не столь очевиден. Продолжим обсуждать отношение **`employee`**, в котором теперь появился атрибут **`phone number`**.

```
Employee:
  • First name
  • Middle name
  • Last name
  • Phone number
```

Является ли телефонный номер независимой сущностью реального мира, или же его можно считать «свойством сотрудника»? Ответ на этот вопрос зависит исключительно от требований предметной области: если для каждого сотрудника по правилам фиксируется только один (не более одного) телефонный номер — всё в порядке, аномалий нет; если для каждого сотрудника может быть зафиксировано несколько телефонных номеров и/или один номер может быть использован несколькими сотрудниками, стоит создать отдельное отношение следующим образом (и установить между ними связь «один ко многим{57}» или «многие ко многим{60}» в процессе дальнейшего проектирования базы данных).

```
Employee:
  • First name
  • Middle name
  • Last name

Phone:
  • Phone number
```

Легко заметить, что хорошей подсказкой здесь является количество экземпляров: если «чего-то» у некой сущности может быть 0 или 1 — это «что-то» можно считать её свойством (например: дата рождения, номер водительского удостоверения, трудовой стаж), если же «чего-то» у некоей сущности может быть больше 1 — это «что-то» стоит вынести в отдельное отношение (например: список детей, список автомобилей, список профессиональных навыков).

Обязательно изучайте требования и ограничения предметной области! Не полагайтесь на интуитивные представления. Например, «интуитивно понятно», что у человека может быть только одна дата рождения, один паспорт и т.д., но теперь представьте, что вы проектируете базу данных разыскиваемых преступников, каждый из которых может скрываться под разными именами, иметь несколько поддельных паспортов и т.д.

И есть ещё одно важное исключение из правила, предписывающего не размещать в одном отношении информацию о нескольких независимых сущностях реального мира — схемы «звезда[120]» и «снежинка[121]». Подробное описание таких схем относится к области хранилищ данных и выходит за рамки данной книги, но небольшой пример мы рассмотрим.

Для начала — графическое пояснение происхождения названия схем (см. рисунок 3.1.e).

Центральная таблица (т.н. «таблица фактов») связана с окружающими её таблицами, содержащими описательную информацию предметной области. В схеме «снежинка» в силу дополнительной нормализации такие таблицы могут в свою очередь ссылаться на другие таблицы, те — на другие и так далее.

В таблице фактов будет собрано большое количество внешних ключей{47} из окружающих её таблиц, и такая ситуация может казаться подозрительной, но на практике здесь не только нет проблем, но даже напротив — таблица фактов может быть очень хорошо нормализована.

---

120 См. подробное описание в статье «The Star Schema» (by Emil Drkušić) [http://www.vertabelo.com/blog/technical-articles/data-warehouse-modeling-the-star-schema]

121 См. подробное описание в статье «The Snowflake Schema» (by Emil Drkušić) [http://www.vertabelo.com/blog/technical-articles/data-warehouse-modeling-the-snowflake-schema]

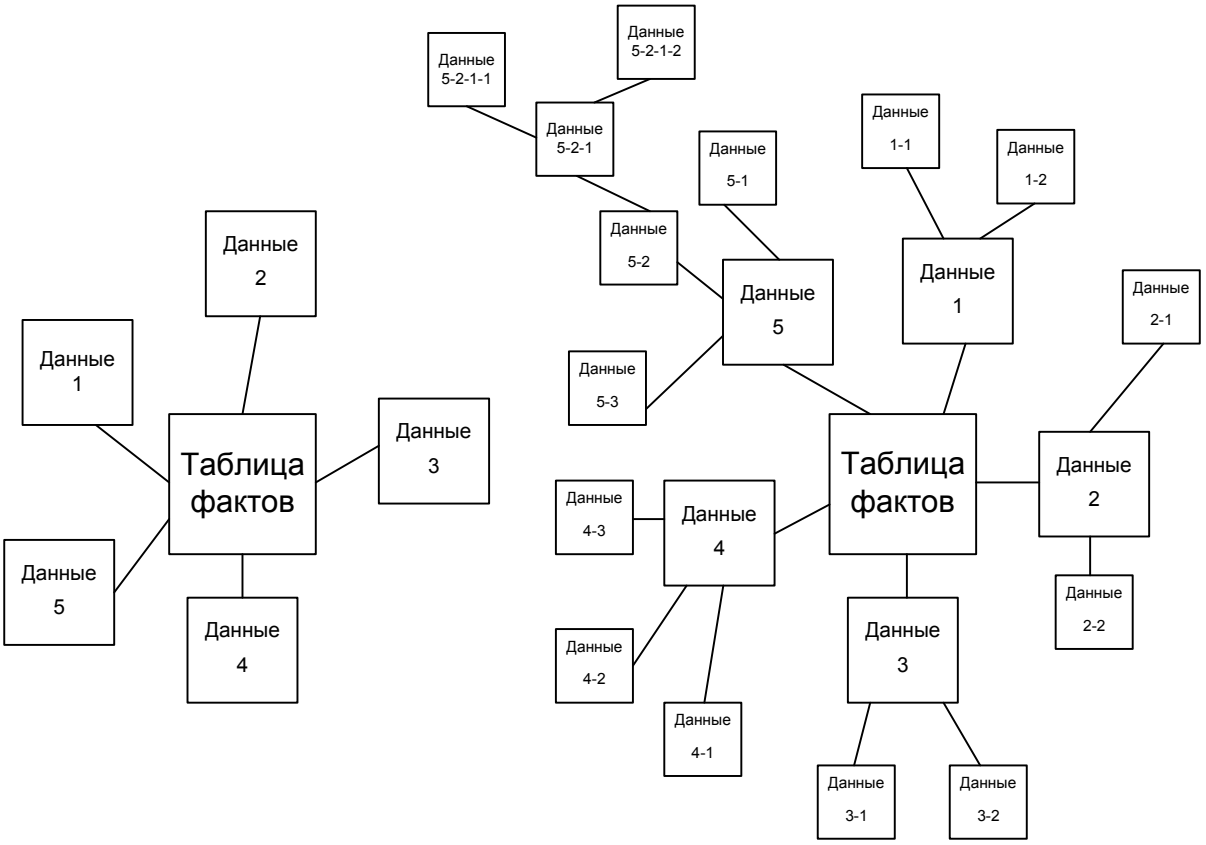


*Рисунок 3.1.e — **Схемы «звезда» (слева) и «снежинка» (справа)***

Для финального закрепления понимания логики схемы «звезда» рассмотрим ещё один (классический) пример (см. рисунок 3.1.f).

Таблицами данных здесь являются `student`, `tutor`, `subject`, `classes_type`, а таблицей фактов — `education_process`, в которой собирается информация о «фактах проведения занятий».

И несмотря на то, что студентов, преподавателей, предметов и видов занятий может быть много, каждый «факт» (т.е. одна запись таблицы `education_process`) отражает лишь сведения о том, что тогда-то такой-то преподаватель проводил занятия такого-то типа по такому-то предмету с таким-то студентом и поставил в итоге такую-то отметку. Потому для данной таблицы аномалии модификации данных не характерны.

К слову, именно такая схема является решением рассмотренной ранее[13] проблемы нарушения такого свойства базы данных как адекватность предметной области[12].

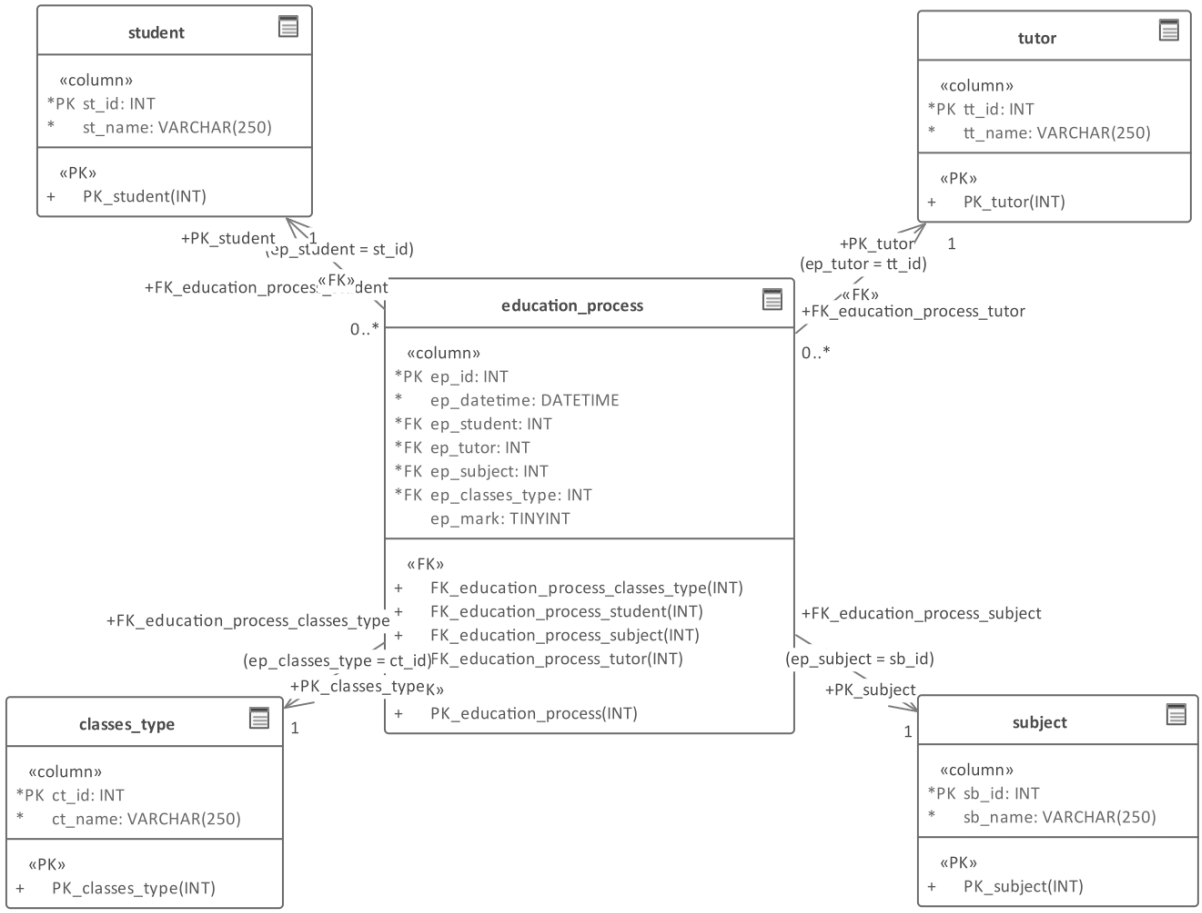


*Рисунок 3.1.f — **Пример схемы «звезда»***

Теперь перейдём к восходящему проектированию{12}, в рамках которого мы получаем отличную возможность продумать реакцию базы данных на выполнение того или иного SQL-запроса.

Здесь можно использовать две основные техники.

Суть первой из них состоит в последовательной проверке каждого запроса на порождение соответствующей аномалии{161}. Т.е. мы должны сознательно искать такие варианты запросов и входных данных, при которых возможно возникновение аномалии. Если поиск увенчался успехом — отлично, мы нашли недостаток в нашей схеме базы данных и можем её переработать.

Если пока у нас нет готового набора запросов (и даже если он есть), прекрасным помощником и генератором идей здесь является представитель заказчика, для которого разрабатывается приложение, использующее нашу базу данных. Он как никто другой знает, какие операции придётся выполнять в процессе его повседневной работы, что особенно ценно, если предметная область является экзотической, и в проектной команде нет людей, хорошо в ней разбирающихся.

Суть второй техники может быть выражена одним словом: тестирование. В самом широком смысле этого слова. Именно в процессе применения различных техник тестирования на разных уровнях проектирования базы данных выявляется множество недостатков, среди которых будут и аномалии операций с данными.

О самом тестировании можно прочитать в соответствующей книге[122], непосредственно тестирование баз данных будет рассмотрено далее{387}, а сейчас мы вернёмся к первой технике и рассмотрим пример исследования запросов на предмет порождения аномалий.

Допустим, у нас есть отношение **`activity`**, представленное на рисунке 3.1.g. Его атрибуты имеют следующее значение:

- **`a_id`** — идентификатор записи (первичный ключ), не имеет «физического смысла», используется для внутренних нужд и гарантированной идентификации записи;
- **`a_employee`** — идентификатор сотрудника (внешний ключ);
- **`a_project`** — имя проекта;
- **`a_role`** — идентификатор роли сотрудника на данном проекте (внешний ключ);
- **`a_start_date`** — дата начала работы сотрудника на данном проекте;
- **`a_end_date`** — дата завершения работы сотрудника на данном проекте.

activity

PK a_id	FK a_employee	a_project	FK a_role	Not NULL a_start_date	a_end_date
115	234	Проект «Дуб»	11	2017.01.09	2017.11.01
116	12	Проект «Ясень»	3	2012.12.21	NULL
119	3134	Проект «Клён»	6	2015.03.01	2016.04.01
121	12	Проект «Дуб»	5	2015.03.15	2016.05.20
...					

*Рисунок 3.1.g — **Отношение `activity` для проведения исследования***

К такому отношению возникает очень много вопросов, но ключевые из них можно сформулировать, продумывая типичные запросы.

Допустим, мы собираемся добавить или изменить запись:

- Как мы можем гарантировать вставку правильного (существующего) названия проекта, если поле **`a_project`** не является внешним ключом?
- Если мы вынуждены приписать сотрудника к проекту тогда, когда дата начала его работы на проекте неизвестна, что мы должны писать в поле **`a_start_date`** (оно объявлено как **`NOT NULL`**, потому придётся указать дату явно)?
- Что будет, если дата завершения работы окажется меньше, чем дата начала работы?
- Что будет, если дата начала и/или завершения работы выйдут за диапазон дат проведения проекта?
- Допустима ли ситуация, при которой один и тот же сотрудник два и более раза упомянут в одной и той же роли на одном и том же проекте в пересекающиеся диапазоны дат?

---

122 «Тестирование программного обеспечения. Базовый курс.» (С. Куликов) [http://svyatoslav.biz/software_testing_book/]

Допустим, мы собираемся удалить запись:

- Если сотрудник увольняется, то верно ли мы понимаем, что придётся удалить все записи о его участии на проектах, где дата завершения работы больше даты увольнения? Или же надо будет изменить дату завершения работы на проекте на дату увольнения?
- Что делать в случае закрытия проекта? Удалять все записи с его упоминанием? Или менять даты завершения работы для всех сотрудников на этом проекте на дату закрытия проекта?

И так далее. Такие вопросы можно задавать очень долго, и лишь ответы представителя заказчика помогут переработать имеющуюся схему базы данных наиболее оптимальным (и при том корректным) образом.

Но даже без таких ответов становится очевидно, что в текущем виде отношение подвержено множеству аномалий и потенциально опасных ситуаций, часть которых не может быть решена даже переработкой схемы базы данных: например, вопрос о выходе дат периода работы сотрудника на проекте за период дат проведения самого проекта должен решаться с помощью триггеров{350}.

Ещё раз подчеркнём самое важное: на любом этапе проектирования стоит постоянно задавать вопросы вида «а если…» и «а верно ли я понимаю, что…» Иногда самое появление такого вопроса позволит заметить недоработку в схеме базы данных, а полноценный ответ и вовсе может показать совершенно иной более эффективный способ хранения и обработки данных в имеющейся ситуации.

И теперь мы переходим к более формальному способу получения отношений, не подверженных аномалиям модификации данных, — к нормализации. В большинстве книг материал нескольких следующих глав излагается в смешанном виде (например, описание процесса нормализации прерывается для рассказа о той или иной зависимости или нормальной форме), но в данном случае для создания более чёткой картины каждая тема будет рассмотрена целостно и непрерывно, а её взаимосвязь с сопутствующими темами будет показана ссылками в уже привычном виде.

Задание 3.1.d. Перед вами — описание отношения file, используемого в базе данных файлообменного сервиса. Исследуйте это отношение на предмет наличия в нём аномалий операций с данными, предложите способы модификации этого отношения с целью устранения аномалий.

Имя поля	Тип данных	Свойства поля	Описание
f_uid	BIGINT UNSIGNED	Первичный ключ.	Глобальный идентификатор файла.
f_fc_uid	BIGINT UNSIGNED	Внешний ключ, NULL-значения разрешены.	Идентификатор категории файлов (ВК на таблицу категорий).
f_size	BIGINT UNSIGNED	NULL-значения запрещены.	Размер файла (в байтах).
f_upload_dt	INTEGER	NULL-значения запрещены.	Дата-время загрузки файла на сервер (в Unixtime).
f_save_dt	INTEGER	NULL-значения запрещены.	Дата-время, до которых файл хранится на сервере (в Unixtime).
f_src_name	VARCHAR(255)	NULL-значения запрещены.	Исходное имя файла (как он назывался у пользователя на компьютере). Без расширения, т.к. оно хранится отдельно в поле f_src_ext!
f_src_ext	VARCHAR(255)	NULL-значения разрешены.	Исходное расширение файла (каким оно было у пользователя на компьютере).
f_name	CHAR(200)	NULL-значения запрещены.	Имя файла на сервере. Пять SHA1-хешей.
f_sha1_cs	CHAR(40)	NULL-значения запрещены.	Контрольная сумма файла, SHA1-хеш.
f_ar_uid	BIGINT UNSIGNED	Внешний ключ, NULL-значения разрешены.	Права доступа к файлу (ВК на таблицу прав доступа).
f_downloaded	BIGINT UNSIGNED	NULL-значения разрешены.	Счётчик скачиваний файла.
f_al_uid	BIGINT UNSIGNED	Внешний ключ, NULL-значения разрешены.	Возрастные ограничения на доступ к файлу (ВК на таблицу возрастных ограничений). Если возрастные ограничения заданы **и** для файла, **и** для категории, к которой принадлежит файл, применяются более строгие ограничения (т.е. берётся максимальный возраст, например: «с 16 лет» и «с 18 лет» — берётся «с 18 лет»).
f_del_link_hash	CHAR(200)	NULL-значения запрещены.	Ссылка на удаление файла (пять SHA1-хешей), если его размещал незарегистрированный пользователь. Зарегистрированные могут удалить файл в своём «личном кабинете».

Задание 3.1.e: стоит ли в базе данных «Банк»{408} использовать схемы{167} «звезда» и «снежинка»? Создайте альтернативный вариант схемы и проверьте, какие аномалии{161} работы с данными появились, а какие исчезли (в сравнении с исходной схемой).

Задание 3.1.f: существуют ли в базе данных «Банк»{408} отношения, в которых, в одно отношение попадает информация о нескольких независимых сущностях реального мира{166}?. Если вы считаете, что «да», доработайте модель так, чтобы устранить этот недостаток.

3.2. ОБЩИЕ ВОПРОСЫ НОРМАЛИЗАЦИИ

3.2.1. ТЕОРИЯ ЗАВИСИМОСТЕЙ

Если вы в достаточной мере знакомы с математическими основами нормализации и просто хотите освежить в памяти сведения о самих нормальных формах, можете сразу перейти к соответствующему разделу{240} или пропустить данную главу и перейти к следующей{211}, в которой будет рассказано о требованиях нормализации.

Материал данной главы будет одним из наиболее теоретических и абстрактных, и в то же время он абсолютно необходим для понимания сути нормальных форм (в определении большинства которых будет фигурировать упоминание той или иной зависимости).

А сами нормальные формы и нормализация, как уже было подчёркнуто ранее{166}, являются одним из наиболее оптимальных способов устранения аномалий операций с данными{161}.

Таким образом, пусть поначалу приведённый далее материал и может показаться «оторванным от жизни», вы всё больше и больше будете понимать его ценность по мере получения практического опыта проектирования баз данных.

Критически важно уже сейчас понять следующий факт: любую из показанных далее зависимостей следует рассматривать строго в контексте предметной области и соответствующих ограничений, т.к. при их изменении меняется и сама зависимость.

Типичной ошибкой начинающих разработчиков баз данных является попытка представить зависимости как некую универсальную абстракцию, справедливую для всех случаев её применения.

Это стремление можно понять, но оно сильно осложняет восприятие концепции зависимостей в ситуации, когда одна и та же переменная отношения при изменении ограничений предметной области то оказывается находящейся в некоторой нормальной форме, то нарушает её (поскольку в ней то есть та или иная зависимость, то её нет).

Потому далее для каждой зависимости будет приведено подробное пояснение случаев её наличия и отсутствия.

Основная терминология теории множеств

Множество (set[123]) — неупорядоченный набор уникальных однотипных элементов, для которого существует операция определения вхождения любого из элементов в данный набор.

Упрощённо: набор элементов одного типа; эти элементы не повторяются (нет дубликатов); всегда можно выяснить, входит некоторый конкретный элемент в набор, или нет.

Далее мы будем часто говорить о множестве атрибутов отношения, потому рассмотрим соответствующий пример (см. рисунок 3.2.а).

---

[123] **Set** — a collection of objects, called elements, with the property that given an arbitrary object x, it can be determined whether or not x appears in the collection (set membership). («The New Relational Database Dictionary», C.J. Date)

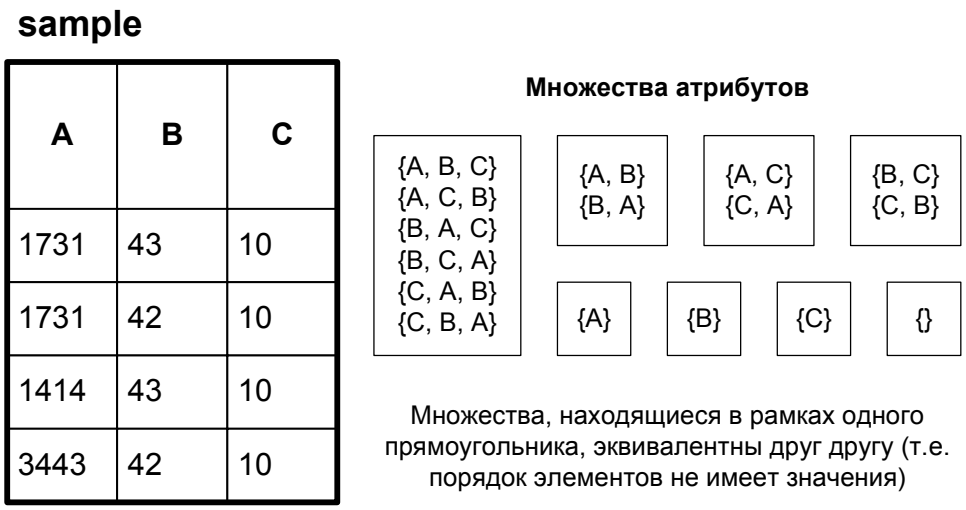


*Рисунок 3.2.а — **Отношение и множества атрибутов***

В математике существует ряд общепринятых символов, относящихся к множествам и операциям над ними. Ключевые из этих символов представлены в следующей таблице.

В математике принято элементы множеств обозначать маленькими буквами, но в теории баз данных (как будет показано далее) этой традиции нет, т.е. как сами множества, так и их элементы будут обозначаться большими буквами.

Символ	Значение	Описание	Пример
Скобки { и }	Множество	Перечень элементов	X = {a, b, c, d, e} Y = {c, e, f, g} Z = {b, c, a, d, e} K = {a, b, c, d} L = {h, i, j} * Эти значения используются в остальных примерах ниже.
∅ или {}	Пустое множество	Множество, не содержащее ни одного элемента	M = {} N = ∅
∈	Элемент	Элемент входит в множество	a ∈ X
∉	Не элемент	Элемент **не** входит в множество	a ∉ Y
| и | или знак #	Мощность	Количество элементов в множестве	|X| = 5 #Y = 4
=	Равенство	Множества содержат одинаковые наборы элементов	X = Z
∩	Пересечение	Набор только таких элементов, которые есть **и** в одном множестве, **и** в другом	X ∩ Y = {c, e}
∪	Объединение	Совокупность элементов их обоих множеств	X ∪ Y = {a, b, c, d, e, f, g}
⊆	Подмножество	Одно множество входит в состав другого	K ⊆ X Z ⊆ X
⊂	Строгое подмножество	Одно множество входит в состав другого, при этом они не равны друг другу	K ⊂ X

$\not\subset$	Не подмножество	Одно множество **не** входит в состав другого	$L \not\subset X$
$\supseteq$	Надмножество	Одно множество содержит в себе другое	$X \supseteq K$ $X \supseteq Z$
$\supset$	Строгое надмножество	Одно множество содержит в себе другое, при этом они не равны друг другу	$X \supset K$
$\not\supset$	Не надмножество	Одно множество не содержит в себе другое	$X \not\supset L$
\ или -	Дополнение (разность)	Набор только таких элементов, которые есть **только** в одном множестве (т.е. их **нет** в другом)	X - Y = {a, b, d}
×	Декартово произведение	Набор всех возможных комбинаций элементов одного и второго множества	Пусть P = {a, b} и Q = {c, d, e}, тогда P × Q = {{a, c}, {a, d), {a, e}, {b, c}, {b, d}, {b, e}}

Поскольку теория множеств несравненно более обширна и сложна, чем показано в этом кратком справочном материале, рекомендуется почитать книгу «The Joy of Sets (2nd ed.)» (Keith Devlin), в которой в очень простой и наглядной форме представлен весь необходимый объём информации.

Основная терминология, относящаяся ко всем видам зависимостей

Существует несколько базовых понятий, как лежащих в основе процесса нормализации, так и используемых в определениях рассмотренных далее зависимостей и нормальных форм. Рассмотрим эти понятия.

Проекция отношения (projection[124]) — подмножество атрибутов отношения и соответствующих им кортежей таких, что все комбинации значений выбранных атрибутов в кортежах проекции встречаются и в исходном отношений.

Упрощённо: из исходного отношения выбирается часть его атрибутов (остальные игнорируются) и все кортежи; в получившемся новом отношении устраняются дубликаты всех кортежей.

Суть проекции проще всего пояснить на знакомом всем школы примере из геометрии (см. рисунок 3.2.b). Допустим, в трёхмерном пространстве положения точек заданы тремя координатами (x, y, z).

Проекция любой точки на каждую из трёх плоскостей будет задана только двумя координатами (например, (x, y)) — данный эффект является аналогом выбора подмножества атрибутов отношения.

Поскольку у нескольких точек в пространстве некая координата может совпадать, на проекции они превратятся в одну точку — данный эффект является аналогом устранения кортежей-дубликатов.

Так на рисунке 3.2.b заданы точки $A(x_1, y_1, z_1)$ и $B(x_2, y_1, z_1)$. Поскольку их координаты y_1 и z_1 совпадают, проекции обеих точек на плоскость YZ превращается в одну точку (совпадают).

---

[124] Let relation r have attributes called A1, A2, …, An (and possibly others). Then (and only then) the expression r{A1,A2,...,An} denotes the **projection** of r on {A1, A2, …, An}, and it returns the relation with heading {A1,A2,...,An} and body consisting of all tuples t such that there exists a tuple in r that has the same value for attributes A1, A2, ..., An as t does.

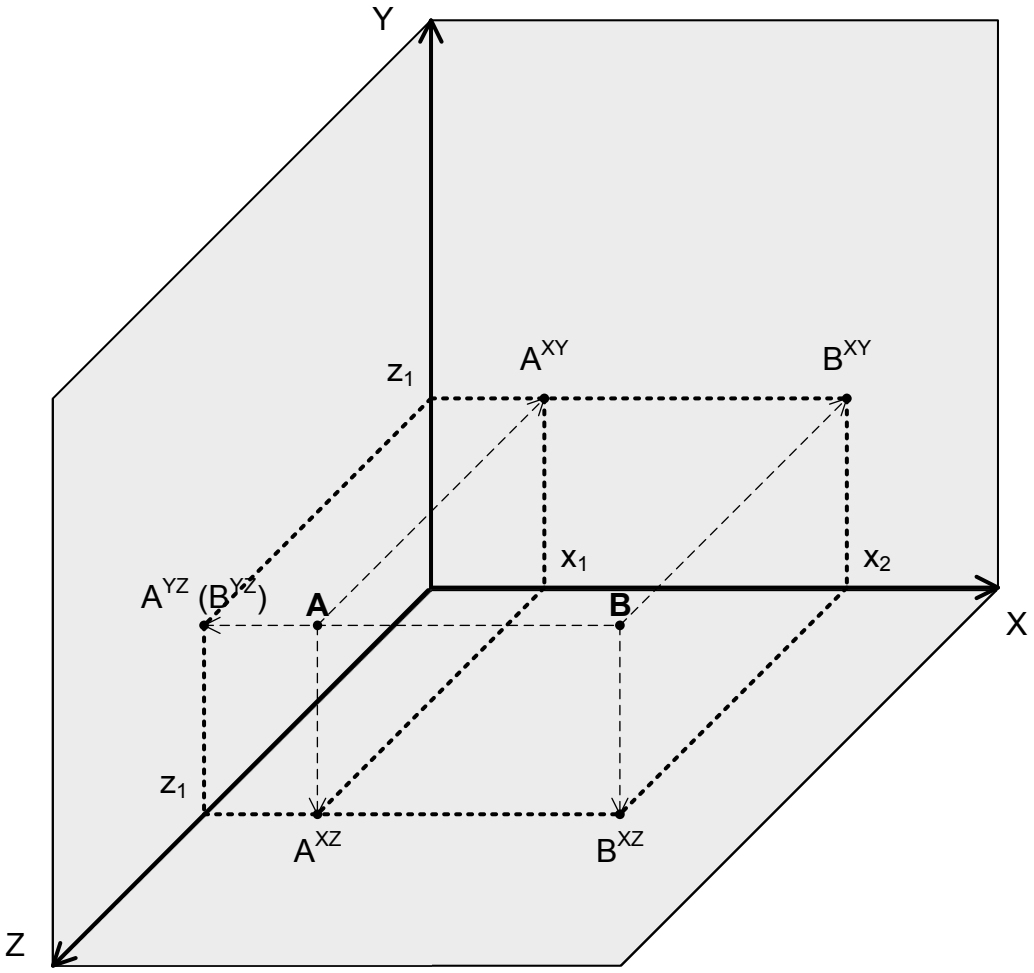


*Рисунок 3.2.b — **Проекции в геометрии***

Теперь от геометрической аналогии вернёмся к базам данных и рассмотрим проекции представленного на рисунке 3.2.a отношения **sample**.

Согласно приведённому выше определению{176} мы выбираем из исходного отношения некий набор атрибутов, а затем в полученном новом отношении устраняем дубликаты.

Для исходного отношения **sample** с атрибутами **A**, **B**, **C** можно получить следующие проекции: **AB**, **BC**, **AC** (показаны на рисунке 3.2.a), а также не имеющие практического смысла проекции **A**, **B**, **C**.

Строго говоря, можно также получить проекцию с нулевым количеством атрибутов (т.е. отношение, в котором нет ничего). Но легко понять, что на практике оно нас тоже не будет интересовать.

Важно понимать, что любое отношение можно спроецировать на любой набор его атрибутов (и чем атрибутов больше, тем больше получается возможных вариантов), однако сейчас будет показано, что далеко не любые проекции оказываются применимыми на практике — ведь из проекций в дальнейшем будет необходимо воссоздать в точности то же самое отношение, которое было изначально.

В приведённом на рисунке 3.2.b примере применимыми оказываются наборы проекций **AB** и **BC**, а также **AB** и **AC**.

*Рисунок 3.2.c — **Проекции в базах данных***

Если же мы возьмём набор проекций **BC** и **AC**, при попытке воссоздать исходное отношение мы получим два лишних кортежа, которых до этого не существовало (см. рисунок 3.2.d). Забегая вперёд, скажем, что для решения этой проблемы существует пятая нормальная форма[268].

Теперь, когда мы подробно рассмотрели понятие проекций отношения, очень легко будет перейти к следующему термину — к декомпозиции без потерь.

В различных источниках можно встретить такие термины как «декомпозиция без потерь», «lossless decomposition», «nonloss decomposition», «lossless-join decomposition», «nonadditive decomposition» — все эти термины обозначают одно и то же (и, как правило, могут быть заменены просто словом «декомпозиция», если из контекста не очевидно, что «декомпозиция без потерь» противопоставляется «просто декомпозиции» (допускающей искажение информации при воссоздании отношения из его проекций)).

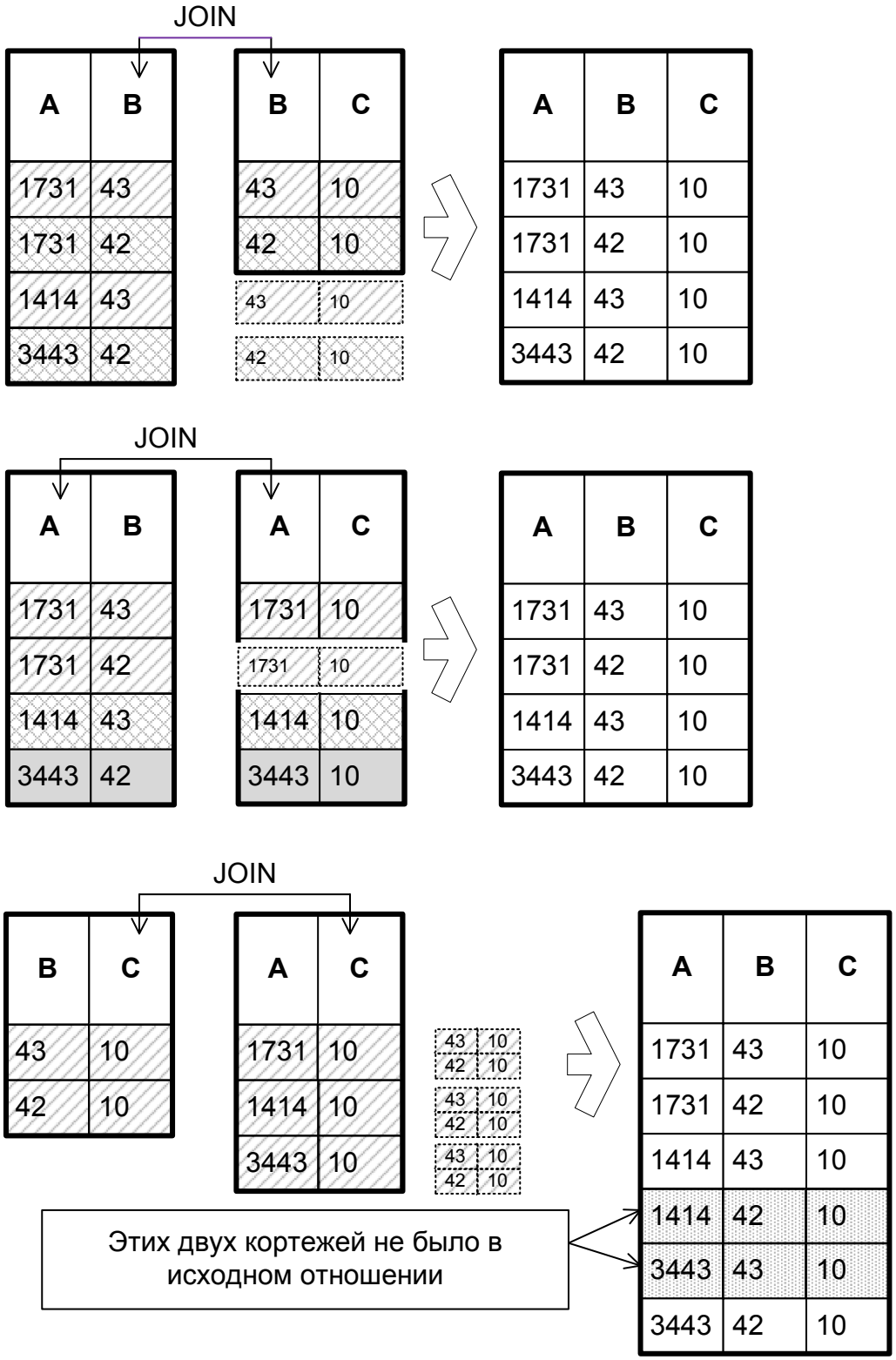


*Рисунок 3.2.d — **Воссоздание исходного отношения из его проекций***

Декомпозиция[125] **без потерь** (nonloss decomposition[126], lossless decomposition, lossless-join decomposition, nonadditive decomposition) — замена отношения набором его проекций, для которого выполняются следующие условия:

- естественное объединение[127] проекций гарантированно приводит к получению в точности исходного отношения (обязательное условие);
- все такие проекции необходимы для получения исходного отношения (необязательное условие);
- как минимум одна из проекций находится в более высокой нормальной форме, чем исходное отношение (необязательное условие).

Упрощённо: исходное отношение разбивается на две и более проекции, причём на их основе можно гарантированно получить исходное отношение, среди нет лишних (ненужных) проекций, как минимум одна из них находится в более высокой нормальной форме.

На рисунке 3.2.c показан плохой случай, т.е. разбиение отношения на такие проекции, среди которых есть лишние, а также не позволяющие получить исходное отношение.

На рисунке 3.2.d показаны два хороших случая, т.е. декомпозиция отношения sample на проекции **AB** + **BC**, **AB** + **AC**, которые в полной мере удовлетворяют всем трём условиям определения: на их основе можно получить исходное отношение, среди них нет лишних проекций, они находятся в более высоких нормальных формах (как минимум проекция BC находится в 5-й нормальной форме{268}).

Теперь, опираясь на только что рассмотренные базовые понятия, мы рассмотрим зависимости, на которых базируются те или иные нормальные формы.

Зависимости, на которых базируется вторая нормальная форма

Да, здесь ничего не пропущено: первая нормальная форма не связана с теорией зависимостей, потому начинаем мы с тех из них, на которых базируется вторая нормальная форма{246}.

Функциональная зависимость (functional dependency[128]) между двумя множествами атрибутов{23} X и Y, являющихся подмножествами атрибутов отношения{21} R, означает наличие *ограничения*, налагаемого на кортежи{23} значения r отношения{24} R, суть которого состоит в том, что при выполнении для двух любых кортежей t_1 и t_2 равенства $t_1[X] = t_2[X]$ также обязано выполняться равенство $t_1[Y] = t_2[Y]$. Функциональная зависимость обозначается X → Y, при этом X называется детерминантом, а Y — зависимой частью[129].

Упрощённо: при наличии функциональной зависимости между множествами атрибутов X и Y существует правило, что если в двух строках таблицы совпадают значения X, то обязаны совпадать и значения Y.

---

125 Термин «декомпозиция» не имеет специального определения в теории баз данных, т.к. является общеупотребимым словом (его синонимы: «разбиение», «разделение», «распределение» и т.д.)

126 **Nonloss decomposition** — replacing a relvar R by certain of its projections R_1, R_2, ..., R_n, such that (a) the join of R_1, R_2, ..., R_n is guaranteed to be equal to R, and usually also such that (b) each of R_1, R_2, ..., R_n is needed in order to provide that guarantee (i.e.none of those projections is redundant in the join), and usually also such that (c) at least one of R_1, R_2, ..., R_n is at a higher level of normalization than R is. («The New Relational Database Dictionary», C.J. Date)

127 Здесь под **естественным объединением** имеется в виду операция NATURAL JOIN, т.е. объединение таблиц по всем их одноимённым столбцам.

128 A **functional dependency**, denoted by X → Y, between two sets of attributes X and Y that are subsets of R specifies a constraint on the possible tuples that can form a relation state r of R. The constraint is that, for any two tuples t_1 and t_2 in r that have $t_1[X] = t_2[X]$, they must also have $t_1[Y] = t_2[Y]$. («Fundamentals of Database Systems», Ramez Elmasri, Shamkant Navathe)

129 Часто «зависимую часть» называют просто «функцией» и говорят «Y — это функция от X».

Для очень быстрого понимания сути функциональной зависимости можно вспомнить школьный курс математики, где часто звучало выражение «Y — это функция от X» (см. рисунок 3.2.е). Действительно, если «множество X» является потенциальным ключом таблицы, этот рисунок в полной мере и без никаких допущений графически иллюстрирует понятие функциональной зависимости.

Важно! Многолетний опыт тренингов показывает, что многие пытаются называть функциональную зависимость «прямой зависимостью». Это неправильно. Более того — эти термины вообще относятся к разным областям науки. Не повторяйте этой ошибки!

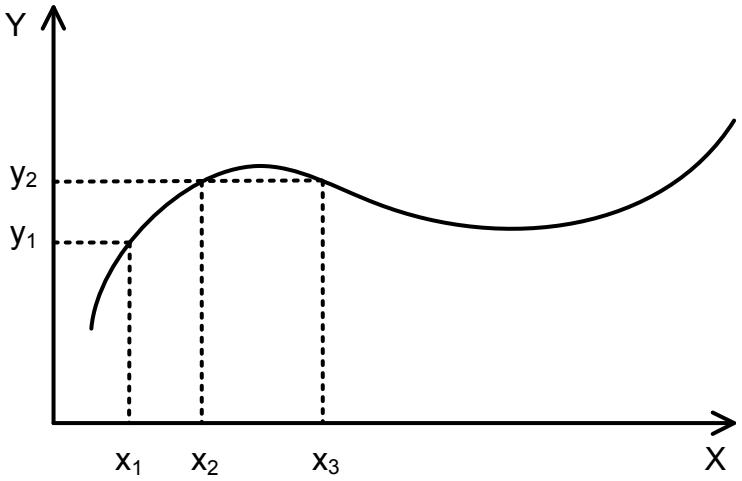


*Рисунок 3.2.е — **Графическое представление функциональной зависимости***

Из определения{180} функциональной зависимости следует, что одному значению множества X должно соответствовать строго одно значение множества Y (действительно, если бы допускалось наличие нескольких различных значений множества Y, могла бы получиться ситуация, при которой в двух и более кортежах находятся одинаковые значения X, но им соответствуют разные значения Y, что противоречит определению).

Это важное следствие вплотную подводит нас к примерам из области баз данных.

Сначала рассмотрим примеры, в которых функциональная зависимость присутствует:

- {Номер паспорта} → {ФИО};
- {Личный номер сотрудника} → {Стаж работы в компании};
- {Личный номер студента, Номер предмета} → {Итоговая оценка}.

Теперь рассмотрим примеры, в которых функциональная зависимость отсутствует:

- {Номер паспорта} → {ФИО};
- {Личный номер сотрудника} → {Стаж работы в компании};
- {Личный номер студента, Номер предмета} → {Итоговая оценка}.

Если вам показалось, что здесь какая-то ошибка, то стоит перечитать предостережение{174} в начале данной главы: до тех пор, пока мы не указали ограничения предметной области, мы не можем гарантировать наличие или отсутствие функциональной зависимости.

Если в рамках предметной области сказано, что одному номеру паспорта соответствует строго одно ФИО, функциональная зависимость {Номер паспорта} → {ФИО} присутствует. Если же сказано, что одному номеру паспорта может соответствовать несколько ФИО (например, на разных языках или с учётом девичей фамилии), функциональной зависимости здесь нет.

Если в рамках предметной области сказано, что одному личному номеру сотрудника соответствует строго одно значение стажа работы в компании, функциональная зависимость {Личный номер сотрудника} → {Стаж работы в компании} присутствует. Если же сказано, что одному личному номеру сотрудника может соответствовать несколько значений стажа работы в компании (например, для случаев увольнения и возвращения сотрудника считается отдельный стаж по каждому периоду работы), функциональной зависимости здесь нет.

Если в рамках предметной области сказано, что для каждого студента по каждому предмету фиксируется только одна итоговая оценка, функциональная зависимость {Личный номер студента, Номер предмета} → {Итоговая оценка} присутствует. Если же сказано, что для каждого студента по каждому предмету может быть несколько итоговых оценок (например, с учётом пересдач экзамена), функциональной зависимости здесь нет.

Ещё раз особо подчеркнём: наличие или отсутствие функциональной зависимости между некими множествами атрибутов определяется исключительно граничениями предметной области.

Очень подробное описание функциональной зависимости и сопутствующих ей терминов можно найти в следующих книгах:

- «Fundamentals of Database Systems (6th edition)» (Ramez Elmasri, Shamkant Navathe) — глава 15.
- «An Introduction to Database Systems (8th edition)» (C.J. Date) — глава 11.

В завершении рассмотрения функциональной зависимости приведём ещё один очень показательный пример, который часто сбивает с толку начинающих разработчиков баз данных. В ранее рассмотренном следствии{181} сказано, что в случае наличия функциональной зависимости X → Y одному значению X соответствует одно значение Y, и это верно. Но обратное — неверно, т.е. одному значению Y вовсе не обязательно должно соответствовать одно значение X.

Проиллюстрируем это на уже рассмотренном примере (см. рисунок 3.2.f) с использованием отношения `result`.

result

PK		
r_student_id FK	**r_subject_id** FK	**r_mark**
1731	43	10
1731	42	10
1414	43	10
3443	42	10

{r_student_id, r_subject_id} → {r_mark}

~~{r_mark} → {r_student_id, r_subject_id}~~

*Рисунок 3.2.f — **Правильное понимание соответствия значений в рамках функциональной зависимости***

Допустим, что в рамках предметной области сказано, что для каждого студента по каждому предмету фиксируется только одна итоговая оценка, т.е. присутствует функциональная зависимость {Личный номер студента, Номер предмета} → {Итоговая оценка}. Но это совершенно не мешает нескольким студентам получить по одному и тому же предмету одинаковые оценки, и никакого нарушения функциональной зависимости здесь нет.

С функциональной зависимостью связано ещё несколько важных терминов, рассмотрение которых необходимо для понимания второй{246} нормальной формы.

Полная функциональная зависимость (full functional dependency[130]) — такая функциональная зависимость X → Y, в которой удаление любого подмножества атрибутов A из множества X приводит к разрушению функциональной зависимости, т.е. для любого A ε X, утверждение (X - A) → Y будет ложным.

Упрощённо: если из множества X удалить хотя бы один элемент, оставшийся набор уже будет недостаточным для выполнения утверждения «одному значению X соответствует одно значение Y».

[130] A functional dependency X → Y is a **full functional dependency** if removal of any attribute A from X means that the dependency does not hold any more; that is, for any attribute A ε X, (X - {A}) does not functionally determine Y. («Fundamentals of Database Systems», Ramez Elmasri, Shamkant Navathe)

Частичная функциональная зависимость (partial functional dependency[131]) — такая функциональная зависимость X → Y, в которой удаление некоторого подмножества атрибутов A из множества X не приводит к разрушению функциональной зависимости, т.е. для некоторого A ε X, утверждение (X - A) → Y будет истинным.

Упрощённо: существует такой элемент, при удалении которого из множества X оставшийся набор всё ещё будет достаточным для выполнения утверждения «одному значению X соответствует одно значение Y».

Эти два определения не зря рассмотрены вместе, т.к. описывают две противоположных ситуации, которые проще всего пояснить на примере составного{40} первичного ключа{39}.

Речь пойдёт именно о составном первичном ключе, т.к. по определению обеих рассматриваемых здесь зависимостей множество X должно допускать удаление хотя бы одного элемента, а в простом{40} первичном ключе изначально находится только один элемент, удаление которого не имеет смысла (т.к., фактически, мы тем самым удалим сам ключ).

Примером полной функциональной зависимости может служить уже рассмотренное ранее отношение **`result`** (см. рисунки 3.2.f, 3.2.g).

Действительно, удаление из составного первичного ключа {**`r_student_id`**, **`r_subject_id`**} любого из полей приведёт к тому, что оценка в поле **`r_mark`** станет относиться либо просто к предмету без учёта студента, либо просто к студенту без учёта предмета. Т.е. мы больше не сможем ответить на вопрос «какую оценку студент такой-то получил по предмету такому-то», т.к. у нас не будет информации либо о студенте, либо о предмете.

Более того, каждое из полей **`r_student_id`** и **`r_subject_id`** по отдельности не может выполнять роль первичного ключа, т.к. в данном отношении значения этих полей объективно должны дублироваться (каждый студент получает оценки по нескольким предметам, следовательно, идентификатор студента должен дублироваться; по каждому предмету оценку получают несколько студентов, следовательно, идентификатор предмета должен дублироваться).

Говоря о частичной функциональной зависимости, стоит подчеркнуть, что в рамках одного отношения может одновременно находиться как несколько полных, так и несколько частичных функциональных зависимостей.

Продемонстрируем это на примере отношения **`result`** (у нас получится одна полная и одна частичная функциональная зависимость, но очевидно, что с добавлением новых атрибутов мы можем получить несколько случаев обеих зависимостей).

---

131 A functional dependency X→Y is a **partial functional dependency** if some attribute A ε X can be removed from X and the dependency still holds; that is, for some A ε X, (X - {A}) → Y. («Fundamentals of Database Systems», Ramez Elmasri, Shamkant Navathe)

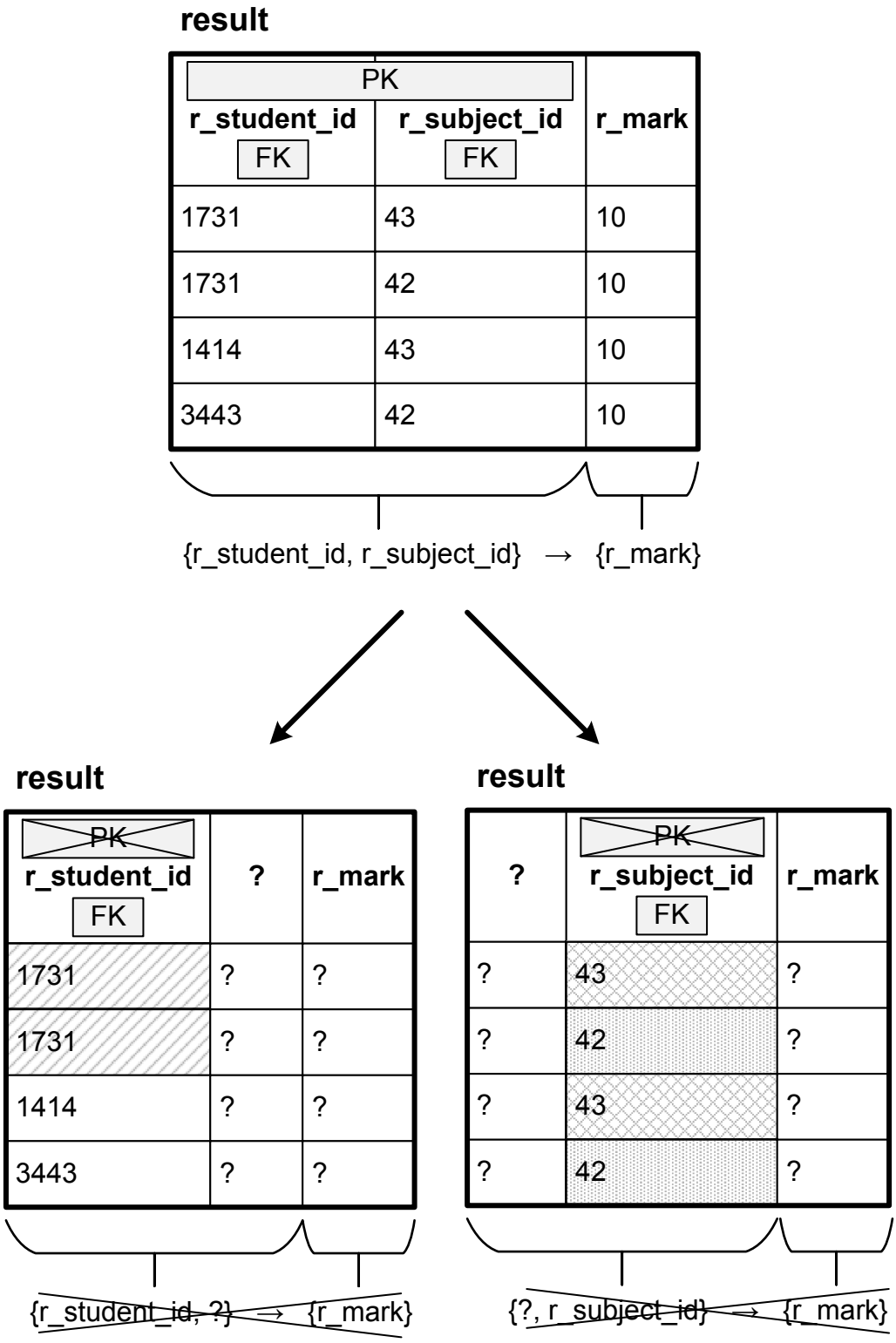


*Рисунок 3.2.g — **Полная функциональная зависимость***

Добавим в отношение `result` атрибут `r_payment_id`, показывающий форму обучения студента (1 = «Бюджет», 2 = «Платно») — см. рисунок 3.2.h. Если вы задумались, не является ли такое решение ошибочным, вы совершенно правы — так мы получаем нарушение второй{246} нормальной формы, но для примера частичной функциональной зависимости такое отношение подходит очень хорошо.

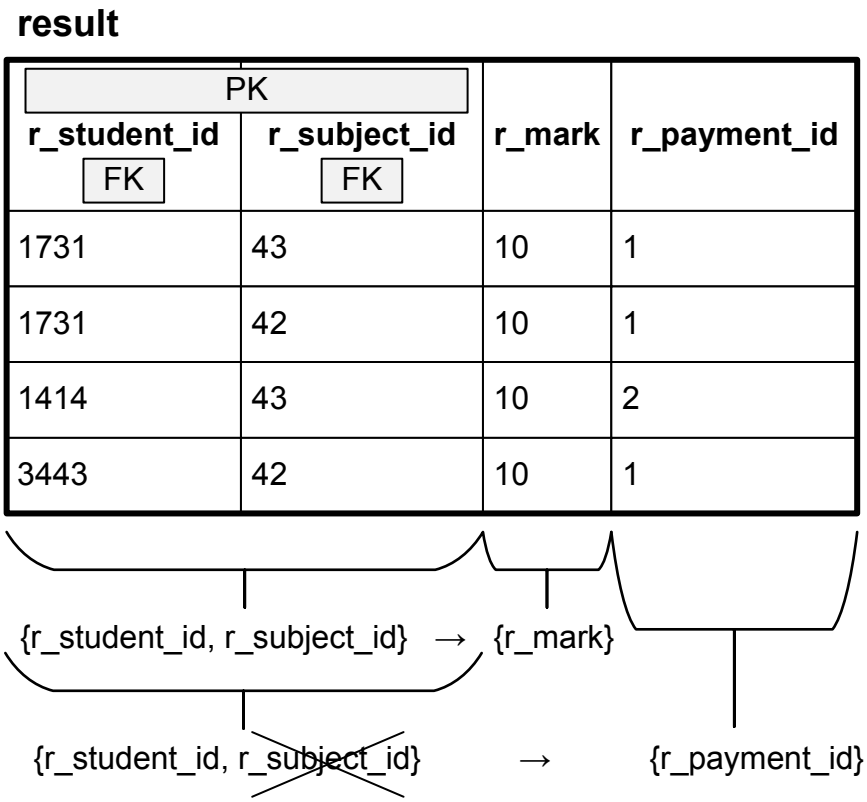


*Рисунок 3.2.h — **Частичная функциональная зависимость***

Очевидно, что значение идентификатора формы обучения студента зависит только от значения идентификатора студента и **не** зависит от значения идентификатора предмета. Потому удаление атрибута `r_subject_id` из набора {`r_student_id`, `r_subject_id`} не приводит к разрушению функциональной зависимости {`r_student_id`, `r_subject_id`} → {`r_payment_id`}.

Для лучшего запоминания ещё раз кратко покажем суть только что рассмотренных зависимостей (см. рисунок 3.2.i):

ФЗ – одному значению X соответствует одно значение Y	$X\{A_1, A_2, \ldots A_n\} \rightarrow Y$
Полная ФЗ – из набора X нельзя удалить ни одного элемента, чтобы не потерять способ выяснения значения Y	$X\{\mathbf{A_1, A_2, \ldots A_n}\} \rightarrow Y$
Частичная ФЗ – из набора X можно удалить некоторые элементы и не потерять способ выяснения значения Y	$X\{\mathbf{A_1, \cancel{A_2}, \ldots A_n}\} \rightarrow Y$

*Рисунок 3.2.i — **Функциональная зависимость, полная функциональная зависимость, частичная функциональная зависимость***

В завершении этой части главы отметим, что функциональная зависимость может быть тривиальной{191} и нетривиальной{191}, что будет рассмотрено далее.

Зависимости, на которых базируются третья нормальная форма и нормальная форма Бойса-Кодда

Следующие разновидности зависимостей (связанные с третьей{252} нормальной формой{258} и нормальной формой Бойса-Кодда{258}) также стоит рассматривать вместе, т.к. они тесно связаны на уровне определений и области применения.

Транзитивная зависимость (transitive dependency[132]) — функциональная зависимость X → Z в отношении R, в которой выполняются условия X → Y и Y → Z, где Y также является подмножеством атрибутов отношения R, но не является потенциальным ключом{37} или подмножеством любого из ключей{35} отношения R.

Упрощённо: «цепочка функциональных зависимостей» вида X → Y → Z, в которой одному значению X соответствует одно значение Y, и одному значению Y соответствует одно значение Z.

Для трёх следующих понятий в заслуживающих доверия официальных первоисточниках не приведено формальных определений, но мы их всё же сформулируем.

Избыточная зависимость (redundant dependency) — зависимость между множествами атрибутов X и Y отношения R, которая может быть выведена на основе других зависимостей внутри отношения R.

Упрощённо: «лишняя» зависимость, которая не даёт дополнительной информации по сравнению с другими зависимостями.

В общем случае избыточная зависимость может быть двух следующих видов:

Избыточная транзитивная зависимость (redundant transitive dependency) — транзитивная зависимость между множествами атрибутов X и Z отношения R (определяемая существованием зависимостей X → Y и Y → Z), в которой также существует зависимость X → Z.

Упрощённо: «цепочка функциональных зависимостей» вида X → Y → Z, в которой существует зависимость X → Z («напрямую», без участия Y).

Избыточная псевдотранзитивная зависимость (pseudotransitive dependency) — избыточная функциональная зависимость, появляющаяся в следующем случае: если для множеств атрибутов X, Y, W, Z отношения R существуют функциональные зависимости X → Y, {Y, W} → Z и {X, W} → Z, то зависимость {X, W} → Z избыточна.

Упрощённо: если детерминант некоторой функциональной зависимости можно вычислить на основе других функциональных зависимостей, нет необходимости в ещё одной функциональной зависимости, детерминант которой «собран» из детерминантов первых двух.

Теперь последовательно покажем все эти случаи на примерах. Начнём с транзитивной зависимости (см. рисунок 3.2.j):

---

132 A functional dependency X → Y in a relation schema R is a **transitive dependency** if there exists a set of attributes Z in R that is neither a candidate key nor a subset of any key of R and both X → Z and Z → Y hold. («Fundamentals of Database Systems», Ramez Elmasri, Shamkant Navathe)

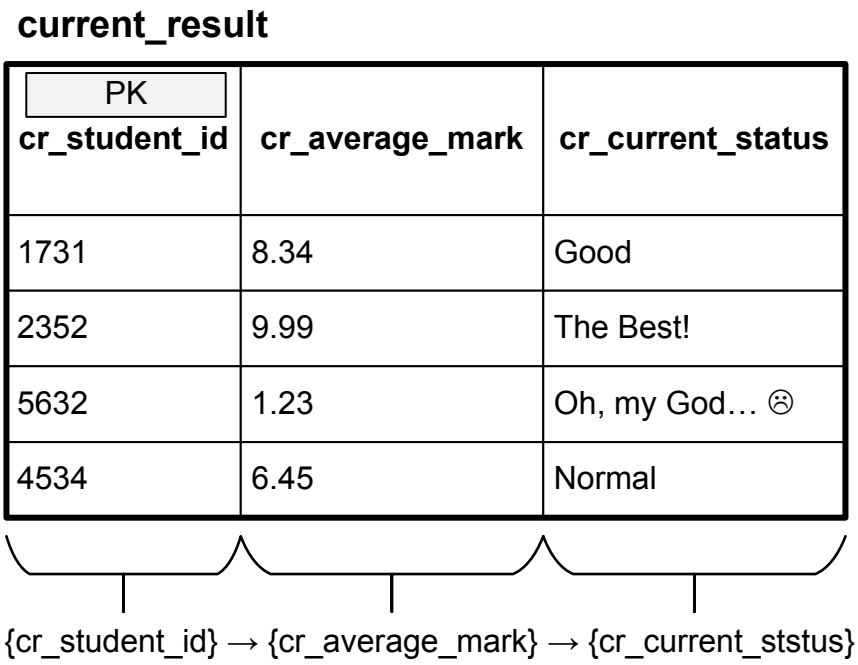


*Рисунок 3.2.j — **Транзитивная зависимость***

Здесь значение среднего балла (атрибут **`cr_average_mark`**) зависит от значения идентификатора студента, а значение статуса (атрибут **`cr_current_status`**) зависит от значения среднего балла (но не зависит от значения идентификатора студента), таким образом мы получаем «цепочку» вида **`Идентификатор студента`** → **`Средний балл`** → **`Статус`**, в которой каждое следующее «звено» зависит от предыдущего — это и есть транзитивная зависимость.

Очевидно, что такая «цепочка» может быть и длиннее трёх элементов, хотя на практике такие случаи встречаются редко.

У избыточной зависимости, как и было отмечено ранее, на практике есть две основных формы проявления — избыточная транзитивная зависимость (см. рисунок 3.2.k) и избыточная псевдотранзитивная зависимость (см. рисунок 3.2.l).

В отношении **`access_control`** значение идентификатора пропуска (атрибут **`ac_pass_id`**) зависит от значения идентификатора студента (атрибут **`ac_student_id`**), а значение имени студента (атрибут **`ac_student_name`**) зависит от значения идентификатора пропуска (атрибут **`ac_pass_id`**). Таким образом, существует транзитивная зависимость **`Идентификатор студента`** → **`Идентификатор пропуска`** → **`Имя студента`**.

Но! Имя студента ведь зависит не только от идентификатора пропуска, но и от идентификатора самого студента, т.е. выполняется условие **`Идентификатор студента`** → **`Имя студента`**. Таким образом, «среднее звено» в этой «цепи» является «лишним» и может быть исключено без потери возможности по идентификатору студента выяснять его имя.

Конечно, если мы просто удалим из отношения **`access_control`** атрибут **`ac_pass_id`**, мы потеряем информацию о пропусках студентов, но в данном случае (как, впрочем, и в других) не идёт речь об удалении атрибутов отношения — мы можем поступить намного более эффективным образом, декомпозировав{180} исходное отношение на несколько новых, о чём подробно будет рассказано при рассмотрении процесса нормализации{223} и самих нормальных форм{240}.

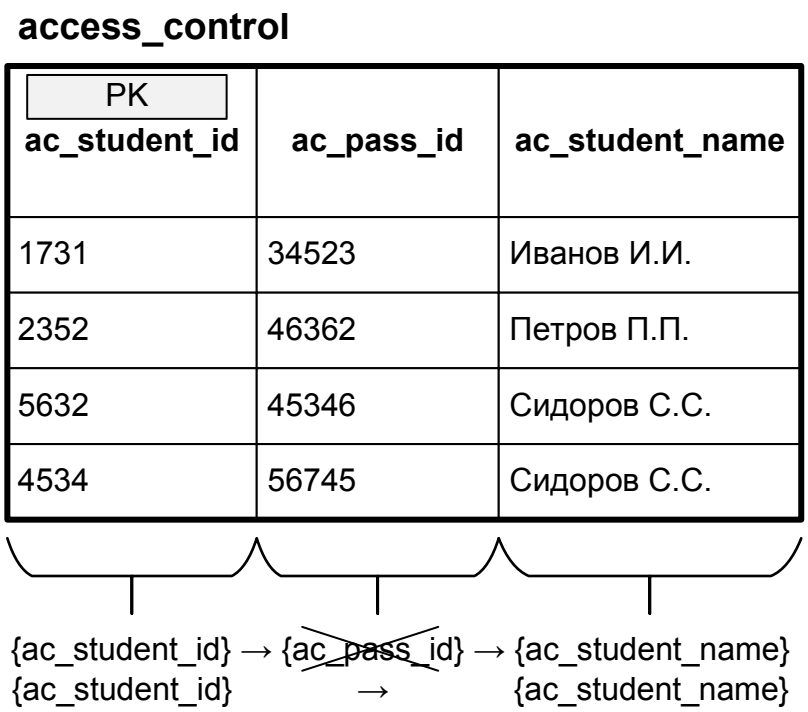


*Рисунок 3.2.k — **Избыточная транзитивная зависимость***

Перед рассмотрением избыточной псевдотранзитивной зависимости в очередной раз напомним: все приведённые здесь рассуждения актуальны только в контексте оговоренных ограничений предметной области.

В данном случае мы немного переработаем отношение **`access_control`** (см. рисунок 3.2.k) и договоримся, что:

- пропуск может быть деактивирован (значение атрибута **`ac_is_pass_active`** будет «Нет») по неким внешним причинам, не зависящим от статуса пропуска (атрибут **`ac_pass_status`**), а зависящим только от идентификатора студента (атрибут **`ac_student_id`**).
- пропуск обязан быть деактивирован (значение атрибута **`ac_is_pass_active`** будет «Нет») в случае, если статус пропуска (атрибут **`ac_pass_status`**) равен «Не выдан» или «Утерян».

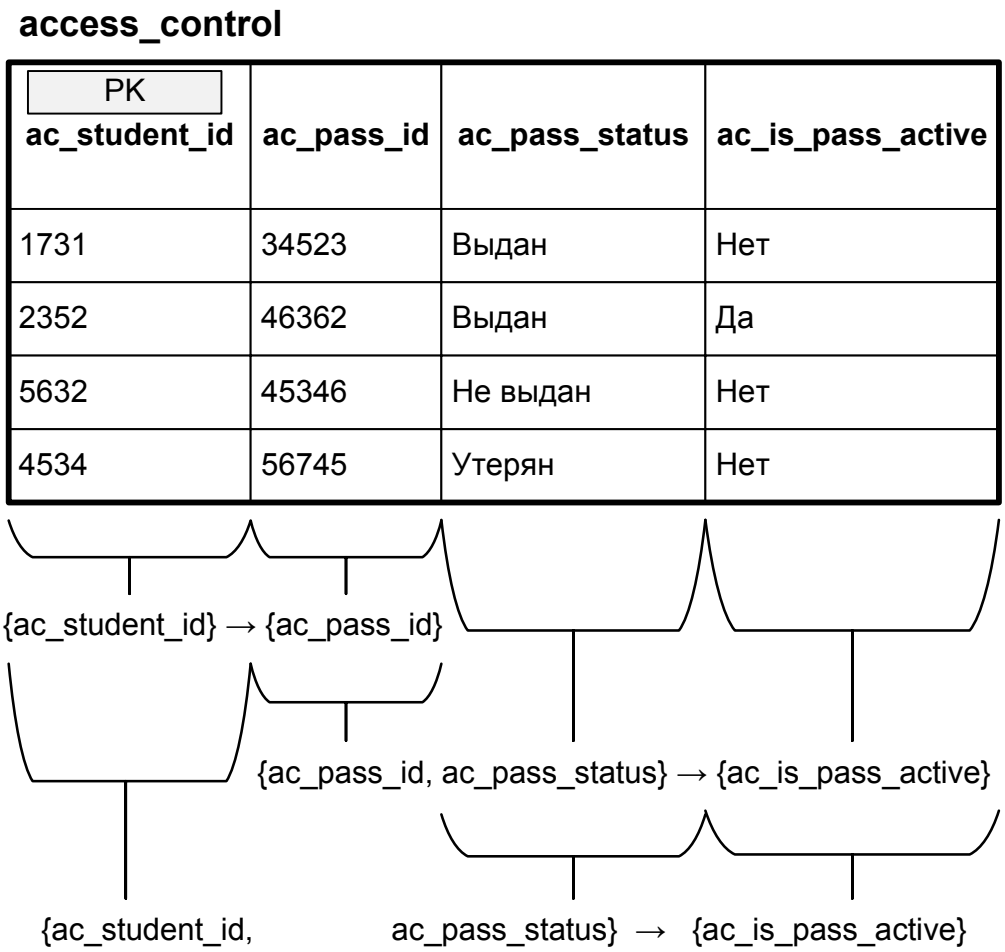


*Рисунок 3.2.l — **Избыточная псевдотранзитивная зависимость***

Таким образом, мы получаем следующие функциональные зависимости:

1. {**ac_student_id**} → {**ac_pass_id**}
2. {**ac_pass_id**, **ac_pass_status**} → {**ac_is_pass_active**}
3. {**ac_student_id**, **ac_pass_status**} → {**ac_is_pass_active**}

Почему третья зависимость является лишней? Потому что, подставив вместо {**ac_student_id**} первую зависимость, мы получим вторую. Это проще пояснить графически — см. рисунок 3.2.m.

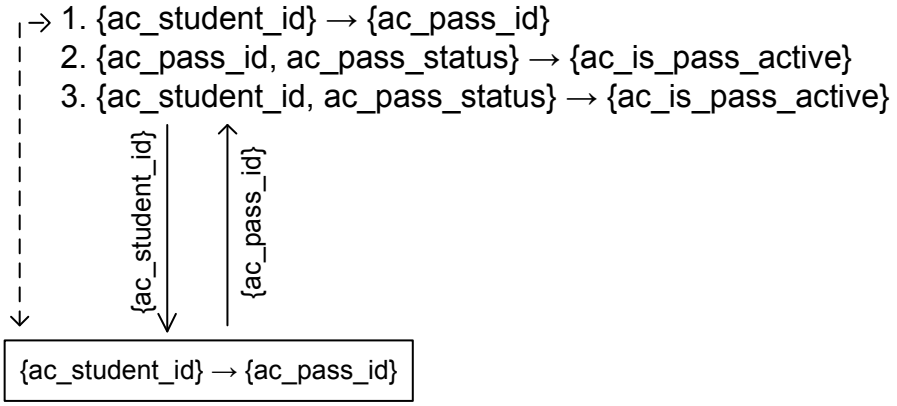


*Рисунок 3.2.m — **Пояснение идеи избыточной псевдотранзитивной зависимости***

Для понимания сути третьей{252} нормальной формы нам нужно рассмотреть ещё два определения функциональных зависимостей.

Тривиальная функциональная зависимость (trivial functional dependency[133]) — функциональная зависимость, которая не может быть нарушена, что выполняется при следующем условии: $X \to Y, X \supseteq Y$.

Упрощённо: зависимость части множества атрибутов от всего множества, по определению не может быть нарушена (т.е. существует всегда).

Нетривиальная функциональная зависимость (nontrivial functional dependency[134]) — функциональная зависимость, которая может быть нарушена, что выполняется при следующем условии: $X \to Y, X \not\supseteq Y$.

Упрощённо: зависимость одного множества атрибутов от другого множества, может как выполняться, так и не выплоняться (нарушаться).

Рассмотрим данные зависимости на графическом примере (рисунок 3.2.n).

Суть тривиальной функциональной зависимости сводится к тому, что если мы знаем целиком весь набор значений множества атрибутов, то мы точно знаем и значение каждого отдельного атрибута (или значение комбинации атрибутов) в этом множестве.

Так, если мы знаем некую пару значений множества {**r_student_id**, **r_subject_id**} (например, {1731, 43}), то мы точно знаем, что в этой паре первый атрибут (**r_student_id**) равен 1731, а второй атрибут (**r_subject_id**) равен 43. Здесь выполняются такие тривиальные функциональные зависимости:

- {**r_student_id**, **r_subject_id**} → {**r_student_id**}
- {**r_student_id**, **r_subject_id**} → {**r_subject_id**}
- {**r_student_id**, **r_subject_id**} → {**r_student_id**, **r_subject_id**}

Ещё один пример, даже чуть более простой: если мы знаем, что ФИО человека {**Фамилия**, **Имя**, **Отчество**} равно {Иванов, Иван, Иванович}, то мы объективно знаем, что его фамилия — Иванов, имя — Иван, отчество — Иванович. Здесь выполняются такие тривиальные функциональные зависимости:

- {**Фамилия**, **Имя**, **Отчество**} → {**Фамилия**}
- {**Фамилия**, **Имя**, **Отчество**} → {**Имя**}
- {**Фамилия**, **Имя**, **Отчество**} → {**Отчество**}
- {**Фамилия**, **Имя**, **Отчество**} → {**Фамилия**, **Имя**}
- {**Фамилия**, **Имя**, **Отчество**} → {**Имя**, **Отчество**}
- {**Фамилия**, **Имя**, **Отчество**} → {**Фамилия**, **Отчество**}
- {**Фамилия**, **Имя**, **Отчество**} → {**Фамилия**, **Имя**, **Отчество**}

И даже ещё более простой пример (для множества, состоящего из одного элемента): если мы знаем, что номер паспорта человека {**Паспорт**} равен AA123456, то мы знаем, что его номер паспорта равен AA123456. Здесь выполняется единственная тривиальная функциональная зависимость:

{**Паспорт**} → {**Паспорт**}

---

133 **Trivial functional dependency** — an FD that can't possibly be violated. The FD X → Y is trivial if and only if X ⊇ Y. («The New Relational Database Dictionary», C.J. Date)

134 **Nontrivial functional dependency** — a functional dependency X→Y is trivial if X ⊇ Y; otherwise, it is **nontrivial**. («Fundamentals of Database Systems», Ramez Elmasri, Shamkant Navathe)

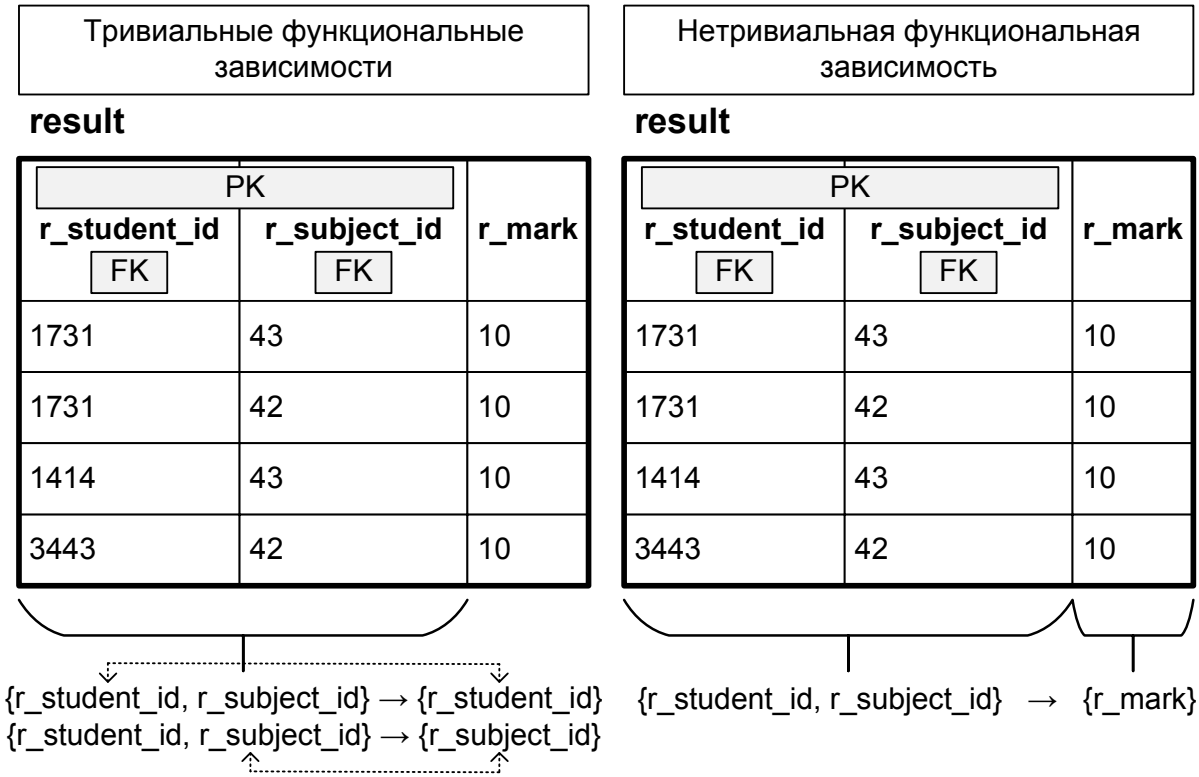


*Рисунок 3.2.n — **Тривиальная и нетривиальная функциональные зависимости***

Нетривиальная функциональная зависимость даже в определении описывается как противоположность тривиальной, т.е. здесь функционально зависимая часть не входит в состав детерминанта функции (так, например, на рисунке 3.2.n атрибут **`r_mark`**, зависящий от множества {**`r_student_id`**, **`r_subject_id`**}, не является частью этого множества).

Рассмотренные ранее полная{183} и частичная{184} функциональные зависимости как раз являются примерами нетривиальных функциональных зависимостей.

Итак, к настоящему моменту мы рассмотрели все зависимости, необходимые для понимания третьей{252} нормальной формы и нормальной формы Бойса-Кодда{258}.

Зависимости, на которых базируется четвёртая нормальная форма

Следующая зависимость представляет собой яркий пример того, как в реляционной теории одни утверждения и выводы влияют на другие: т.н. «многозначная зависимость» является следствием первой нормальной формы{241}, запрещающей создавать в отношениях многозначные атрибуты.

Многозначная зависимость (multivalued dependency[135]) — зависимость в отношении{21} R (обозначаемая $X \twoheadrightarrow Y$ или $X \twoheadrightarrow Y|Z$, где X, Y и $Z = (R — (X \cup Y))$ — подмножества атрибутов{23} R), определяющая следующее ограничение. При наличии двух кортежей{23} t_1 и t_2 таких, что $t_1[X] = t_2[X]$, обязаны также существовать кортежи t_3 и t_4 и выполняться условия:

а) $t_3[X] = t_4[X] = t_1[X] = t_2[X]$;

б) $t_3[Y] = t_1[Y]$ и $t_4[Y] = t_2[Y]$;

в) $t_3[Z] = t_2[Z]$ и $t_4[Z] = t_1[Z]$.

Упрощённо: многозначная зависимость обязывает отношение, содержащее два кортежа, совпадающие по значению одного из трёх атрибутов, содержать также два дополнительных кортежа, «перекрёстно» содержащие комбинации оставшихся двух атрибутов (см. рисунок 3.2.о).

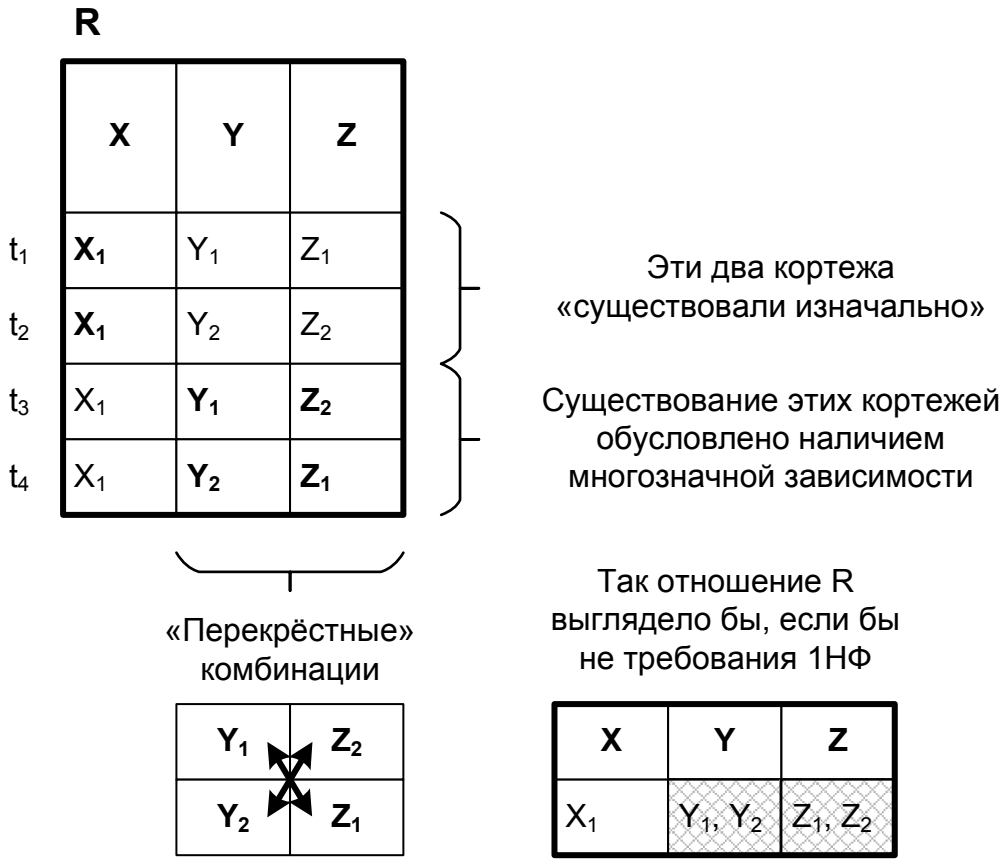


*Рисунок 3.2.о — **Графическое пояснение многозначной зависимости***

---

135 A **multivalued dependency** $X \twoheadrightarrow Y$ specified on relation schema R, where X and Y are both subsets of R, specifies the following constraint on any relation state r of R: If two tuples t_1 and t_2 exist in r such that $t_1[X] = t_2[X]$, then two tuples t_3 and t_4 should also exist in r with the following properties, where we use Z to denote $(R — (X \cup Y))$: a) $t_3[X] = t_4[X] = t_1[X] = t_2[X]$; b) $t_3[Y] = t_1[Y]$ and $t_4[Y] = t_2[Y]$; c) $t_3[Z] = t_2[Z]$ and $t_4[Z] = t_1[Z]$. («Fundamentals of Database Systems», Ramez Elmasri, Shamkant Navathe)

Говоря совсем неформальным языком, можно отметить, что при наличии многозначной зависимости «внутри отношения существуют две связи вида один ко многим», т.е. одному значению X соответствует много (два и более) значений Y, и также одному значению X соответствует много (два и более) значения Z.

Единственным способом отразить такую ситуацию без нарушения 1НФ является добавление дополнительных кортежей, т.к. без них не получилось бы показать независимость значений Y и Z друг от друга (могло бы показаться, что $Y \to Z$, т.е. значению Y_1 строго соответствует Z_1, а Y_2 строго соответствует Z_2).

Это рассуждение приводит нас к двум следующим определениям.

Тривиальная многозначная зависимость (trivial multivalued dependency[136]) — многозначная зависимость $X \twoheadrightarrow Y$, в которой Y является подмножеством X или $R = X \cup Y$.

Упрощённо: многозначная зависимость является тривиальной, если множество атрибутов Y входит в состав множества атрибутов X, или совокупность множеств X и Y составляет весь набор атрибутов отношения R (тогда по основному определению{193} *для множества Z не остаётся атрибутов, т.е. оно является пустым).*

Нетривиальная многозначная зависимость (nontrivial multivalued dependency[136]) — многозначная зависимость $X \twoheadrightarrow Y$, в которой Y не является подмножеством X и $R \neq X \cup Y$.

Упрощённо: многозначная зависимость является нетривиальной, если множество атрибутов Y не входит в состав множества атрибутов X, и совокупность множеств X и Y не составляет весь набор атрибутов отношения R (чтобы по основному определению{193} *для множества Z осталась часть атрибутов, не входящих ни в X, ни в Y).*

Легко догадаться, что при рассмотрении четвёртой нормальной формы{263} нас будет интересовать нетривиальная многозначная зависимость (именно она представлена на рисунке 3.2.о). А оба случая, когда многозначная зависимость является тривиальной, показаны на рисунке 3.2.р.

Существует также ещё один неформальный признак того, что многозначная зависимость является тривиальной: если многозначная зависимость существует для атрибутов X, Y, Z, и при этом «внутри» существует любая функциональная зависимость{180} ($X \to Y$, $X \to Z$, $Y \to Z$, $Y \to X$ и т.д., т.е. любая) — в такой ситуации многозначная зависимость выродится до одного из двух показанных на рисунке 3.2.р случаев.

---

136 An MVD $X \twoheadrightarrow Y$ in R is called a **trivial MVD** if (a) Y is a subset of X, or (b) $X \cup Y = R$. An MVD that satisfies neither (a) nor (b) is called a **nontrivial MVD**. («Fundamentals of Database Systems», Ramez Elmasri, Shamkant Navathe)

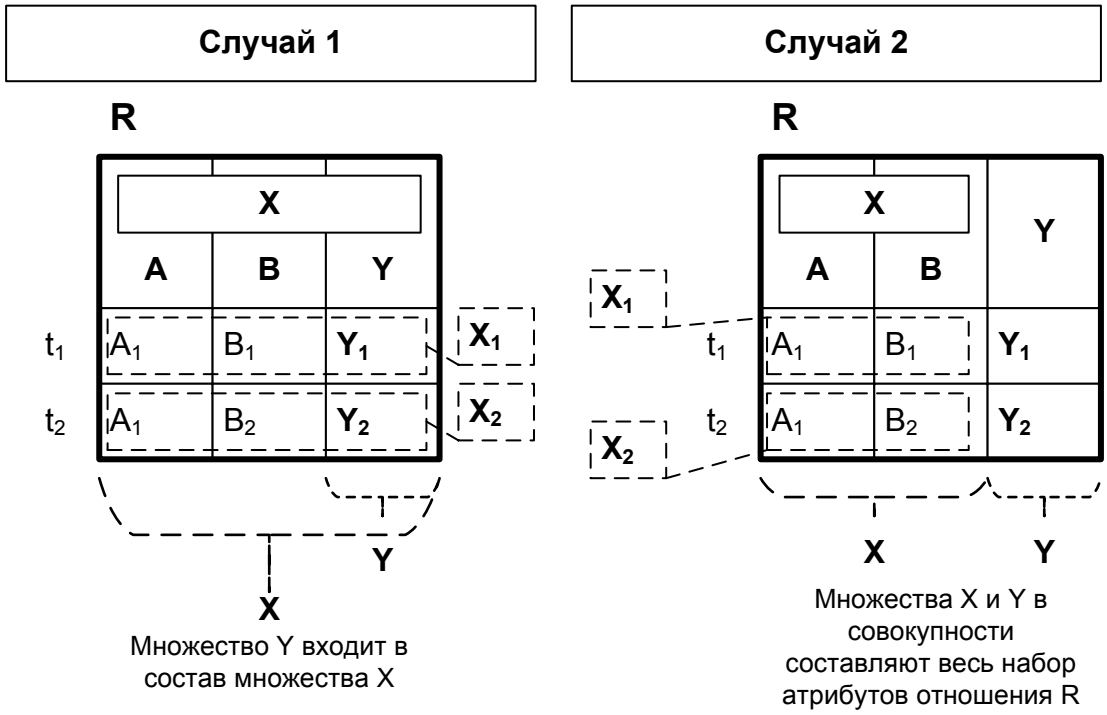


*Рисунок 3.2.р — **Оба случая получения тривиальной многозначной зависимости***

Итак, теперь вы можете временно переключиться на изучение четвёртой нормальной формы{263}, т.к. только что мы рассмотрели положенные в её основу зависимости.

Зависимости, на которых базируется пятая нормальная форма

Пятая нормальная форма{268} базируется на т.н. зависимости соединения, являющейся обобщённым случаем многозначной зависимости{193} (являющейся, в свою очередь, обобщением функциональной зависимости[137]).

Зависимость соединения (join dependency[138]) — зависимость в отношении R, обозначаемая JD(R_1, R_2, ..., R_n) и порождающая следующее ограничение: при любом наборе данных отношение R должно допускать декомпозицию без потерь{180} на проекции{176} R_1, R_2, ..., R_n.

Упрощённо: взаимная зависимость между несколькими (более двух) атрибутами отношения, обязывающая отношение содержать только такие данные, при которых декомпозиция отношения на проекции из групп этих атрибутов будет декомпозицией без потерь (т.е. позволяющей восстановить в точности исходное отношение).

Тривиальная зависимость соединения (trivial join dependency[139]) — такая зависимость соединения JD(R_1, R_2, ..., R_n), в которой хотя бы один компонент R_i содержит в себе полный набор атрибутов отношения R.

Упрощённо: зависимость соединения, которая выполняется всегда (в силу того факта, что отношение всегда можно в точности восстановить «из самого себя» (именно этот эффект достигается, если хотя бы одна из проекций содержит полный набор атрибутов отношения, т.е. является, фактически, самим же исходным отношением)).

Нетривиальная зависимость соединения (nontrivial join dependency[140]) — такая зависимость соединения JD(R_1, R_2, ..., R_n), в которой ни один компонент R_i не содержит в себе полный набор атрибутов отношения R.

Упрощённо: зависимость соединения, которая может не выполняться, т.е. может существовать ситуация, при которой отношение нельзя без искажений восстановить из его проекций на входящие в зависимость группы атрибутов.

Немногие готовы спорить с тем, что определение зависимости соединения звучит достаточно запутанно, потому сразу переходим к примерам, но всё же предварительно добавим небольшое неформальное уточнение: зависимости соединения часто (но не всегда!) проявляются там, где отношение невозможно без потерь декомпозировать на две проекции, но на три и более — возможно.

Начнём с почти детского примера, демонстрирующего зависимость соединения в отношении, в котором все неключевые атрибуты{23} функционально зависят{180} от первичного ключа{39} и при этом никак не зависят друг от друга — такое отношение `elective` (описывающее факультативы) показано на рисунке 3.2.q.

---

137 Этот вопрос подробно освещён в разделе 15.7 книги «Fundamentals of Database Systems, 6th edition» (Ramez Elmasri, Shamkant Navathe).

138 A **join dependency** (JD), denoted by JD(R_1, R_2, ..., R_n), specified on relation schema R, specifies a constraint on the states r of R. The constraint states that every legal state r of R should have a nonadditive join decomposition into R1, R2, …, Rn. («Fundamentals of Database Systems», Ramez Elmasri, Shamkant Navathe)

139 A join dependency JD(R_1, R_2, ..., R_n), specified on relation schema R, is a **trivial JD** if one of the relation schemas R_i in JD(R_1, R_2, ..., R_n) is equal to R. («Fundamentals of Database Systems», Ramez Elmasri, Shamkant Navathe)

140 Данное определение формулируется как обратный случай тривиальной зависимости соединения, потому — см. сноску 139.

Исходное отношение

elective

PK **e_id**	**e_group**	**e_tutor**	**e_topic**
157	1	33	Stop using ORMs!
248	8	24	MySQL vs ...
411	1	78	Oracle 999 overview
489	7	33	Arduino in free space

Функциональные зависимости:
{e_id} → {e_group}
{e_id} → {e_tutor}
{e_id} → {e_topic}

Зависимость соединения:
JD((e_id, e_group),
(e_id, e_tutor),
(e_id, e_topic))

Проекции

P_1 (e_id, e_group)

PK **e_id**	**e_group**
157	1
248	8
411	1
489	7

P_2 (e_id, e_tutor)

PK **e_id**	**e_tutor**
157	33
248	24
411	78
489	33

P_3 (e_id, e_topic)

PK **e_id**	**e_topic**
157	Stop using ORMs!
248	MySQL vs ...
411	Oracle 999 overview
489	Arduino in free space

*Рисунок 3.2.q — **Самый простой пример зависимости соединения***

Этот пример действительно тривиален, т.к. не составляет труда понять, что исходное отношение гарантированно удастся восстановить, если каждая его проекция содержит в себе первичный ключ исходного отношения. Тем не менее, даже такой случай является иллюстрацией идеи зависимости соединения.

Перейдём к более сложному примеру. Рассмотрим отношение **`workload`**, показывающее нагрузку преподавателей на различных факультетах (см. рисунок 3.2.r). Для наглядности оставим в отношении строки, хотя в настоящей базе данных вместо надписей были бы числовые значения, а сами поля являлись бы внешними ключами.

Очевидно, что для такого отношения только представленные на рисунке 3.2.r три проекции имеют смысл (оставшиеся варианты — это ещё три проекции на один атрибут каждая (что не имеет смысла) и проекция на все исходные атрибуты, т.е. само отношение).

Исходное отношение

workload

	PK		
	w_tutor	**w_subject**	**w_faculty**
(1)	Иванов И.И.	Математика	Точных наук
(2)	Иванов И.И.	Информатика	Естествознания
(3)	Петров П.П.	Математика	Естествознания
(4)	Сидоров С.С.	Информатика	Кибернетики
(5)	Петров П.П.	Физика	Точных наук
(6)	Петров П.П.	Математика	Точных наук
(7)	Иванов И.И.	Математика	Естествознания

Функциональные зависимости: нет.

Зависимость соединения:
JD((w_tutor, w_subject),
(w_subject, w_faculty),
(w_tutor, w_faculty))

Отмеченные фоном кортежи появляются **только** в силу наличия правила: «Если преподаватель ведёт некоторый предмет **П**, и на некотором факультете **Ф** преподаётся этот предмет **П**, и преподаватель ведёт хотя бы один любой предмет на этом факультете **Ф**, то он обязан вести на этом факультете **Ф** и предмет **П**.»

а) На факультете «Точных наук» преподаётся «Математика» (1).
б) Преподаватель «Петров П.П.» преподаёт на факультете «Точных наук» (5).
в) Преподаватель «Петров П.П.» преподаёт «Математику» (3).
ЗНАЧИТ: преподаватель «Петров П.П.» должен преподавать «Математику» на факультете «Точных наук» (6).

а) На факультете «Естествознания» преподаётся «Математика» (3).
б) Преподаватель «Иванов И.И.» преподаёт на факультете «Естествознания» (2).
в) Преподаватель «Иванов И.И.» преподаёт «Математику» (1).
ЗНАЧИТ: преподаватель «Иванов И.И.» должен преподавать «Математику» на факультете «Естествознания» (7).

Проекции

P_1 (w_tutor, w_subject)

PK	
w_tutor	**w_subject**
Иванов И.И.	Математика
Иванов И.И.	Информатика
Петров П.П.	Математика
Сидоров С.С.	Информатика
Петров П.П.	Физика

P_2 (w_subject, w_faculty)

PK	
w_subject	**w_faculty**
Математика	Точных наук
Информатика	Естествознания
Математика	Естествознания
Информатика	Кибернетики
Физика	Точных наук

P_3 (w_tutor, w_faculty)

PK	
w_tutor	**w_faculty**
Иванов И.И.	Точных наук
Иванов И.И.	Естествознания
Петров П.П.	Естествознания
Сидоров С.С.	Кибернетики
Петров П.П.	Точных наук

Никаких двух из этих проекций недостаточно, чтобы получить исходное отношение (появляются лишние записи, которых ранее не было). В то время как на основе всех трёх проекций получается в точности исходное отношение.

*Рисунок 3.2.r — **Пример зависимости соединения***

Как и отмечено на рисунке 3.2.r, описанная зависимость соединения актуальна в силу наличия правила (ограничения) предметной области: «Если преподаватель ведёт некоторый предмет **П**, и на некотором факультете **Ф** преподаётся этот предмет **П**, и преподаватель ведёт хотя бы один любой предмет на этом факультете **Ф**, то он обязан вести на этом факультете **Ф** и предмет **П**.».

Вы можете самостоятельно убедиться (с использованием любой СУБД, поддерживающей операцию **`NATURAL JOIN`**), что объединение любых двух из представленных на рисунке 3.2.r проекций порождает лишние атрибуты, и лишь объединение всех трёх проекций позволяет получить в точности исходное отношение.

На всякий случай приведём SQL-код для получения корректного результата (в синтаксисте MySQL).

MS SQL **Восстановление исходного отношения по трём проекциям**

```
SELECT *
FROM   (SELECT *
        FROM   `p1`
               NATURAL JOIN `p2`) AS `p12`
       NATURAL JOIN `p3`
```

Но что будет, если «отменить» это сложное и запутанное правило о преподавателях, предметах, факультетах и их взаимосвязи? Получится результат, представленный на рисунке 3.2.s.

Обратите внимание, что проекции P_1, P_2, P_3 на обоих рисунках (3.2.r и 3.2.s) совершенно одинаковы, в то время как отношения, показанные на этих рисунках, различаются. Чуть выше приведён SQL-код, выполнение которого наглядно демонстрирует, что в результате объединения полученных проекций получается отношение, представленное на рисунке 3.2.r, т.е. **не** получается отношение, представленное на рисунке 3.2.s.

Таким образом, устранение правила, порождающего зависимость соединения, привело к получению отношения, декомпозиция которого без потерь невозможна.

Исходное отношение

workload

PK		
w_tutor	**w_subject**	**w_faculty**
Иванов И.И.	Математика	Точных наук
Иванов И.И.	Информатика	Естествознания
Петров П.П.	Математика	Естествознания
Сидоров С.С.	Информатика	Кибернетики
Петров П.П.	Физика	Точных наук

Функциональные зависимости: нет.

Зависимости соединения: нет.

Проекции

P_1 (w_tutor, w_subject)

PK	
w_tutor	**w_subject**
Иванов И.И.	Математика
Иванов И.И.	Информатика
Петров П.П.	Математика
Сидоров С.С.	Информатика
Петров П.П.	Физика

P_2 (w_subject, w_faculty)

PK	
w_subject	**w_faculty**
Математика	Точных наук
Информатика	Естествознания
Математика	Естествознания
Информатика	Кибернетики
Физика	Точных наук

P_3 (w_tutor, w_faculty)

PK	
w_tutor	**w_faculty**
Иванов И.И.	Точных наук
Иванов И.И.	Естествознания
Петров П.П.	Естествознания
Сидоров С.С.	Кибернетики
Петров П.П.	Точных наук

Никакие комбинации представленных проекций не позволяют получить в точности исходное отношение.

*Рисунок 3.2.s — **Пример отсутствия зависимости соединения***

Итак, теперь вы можете временно переключиться на изучение пятой нормальной формы{268}, т.к. только что мы рассмотрели положенные в её основу зависимости, и даже почти сформулировали её определение.

Зависимости, на которых базируется доменно-ключевая нормальная форма

Представленные ниже зависимости никак не связаны с рассмотренными ранее. Они были специально сформулированы Рональдом Фагином (автором доменно-ключевой нормальной формы) и в целом куда более просты для понимания, чем многие «классические» зависимости.

Доменная зависимость (domain dependency[141]) — зависимость в отношении R, обозначаемая IN(A, S) и порождающая следующее ограничение: любое значение атрибута A должно быть членом множества S.

Упрощённо: представим множество «день недели» (ПН, ВТ, СР, ЧТ, ПТ, СБ, ВС) и атрибут некоего отношения R, названный `day_of_week` *и относящийся к домену «день недели»; ограничение полученной доменной зависимости выполняется только в том случае, если ни в одной записи таблицы в поле* `day_of_week` *нет никаких иных значений, кроме ПН, ВТ, СР, ЧТ, ПТ, СБ, ВС.*

Ещё более упрощённо: значение любого поля таблицы должно соответствовать перечню (или диапазону) допустимых для него значений.

Ключевая зависимость (key dependency[142]) — зависимость в отношении R, обозначаемая KEY(K) и порождающая следующее ограничение: никакие два кортежа отношения R не могут иметь одинаковых значений множества K.

Упрощённо: если некое множество K полей таблицы является ключом{35}, то в каждой строке таблицы совокупность значений атрибутов, входящих в состав K, уникальна.

Ещё более упрощённо: значения ключей{35} таблицы не должны дублироваться.

Начнём пояснение с ключевой зависимости, т.к. она вполне самоочевидна: действительно, любая современная СУБД по определению гарантирует уникальность первичных ключей{39} и уникальных индексов{109}. Однако Фагин подчёркивает[143], что в его определении речь идёт скорее о суперключе{36}, т.е. свойство минимальности ключа нас здесь не интересует.

Это уточнение могло бы показать несущественным, если бы оно не переводило наш фокус внимания с конкретной реализации схемы отношения в виде таблицы СУБД на непосредственно саму схему отношения, отражающую особенности предметной области.

С учётом этого замечания «ещё более упрощённую» формулировку ключевой зависимости можно сформулировать следующим образом: если согласно правилам предметной области совокупность значений неких полей отношения должна быть уникальная для любой записи, то ключевая зависимость присутствует в отношений тогда и только тогда, когда это ограничение соблюдено (вне зависимости от того, используются ли первичный ключ, уникальный индекс или какие бы то ни было иные технические средства реализации данного ограничения).

---

141 The **domain dependency** IN(A,S), where A is an attribute and S is a set, means that the A entry in each tuple must be a member of the set S. For example, let A be the attribute SALARY, and let S be the set of all integers between 10'000 and 100'000. If A is one of the attributes of relation R, then R obeys IN(A, S) if and only if the SALARY entry of every tuple of R is an integer between 10'000 and 100'000. («A Normal Form for Relational Databases That Is Based on Domains and Keys», Ronald Fagin)

142 The **key dependency** KEY(K), where K is a set of attributes, says that K is a key, that is, that no two tuples have the same K entries. («A Normal Form for Relational Databases That Is Based on Domains and Keys», Ronald Fagin)

143 We remark that keys are usually defined at the schema level, not at the instance level. At the schema level, for K to be a key, it is usually assumed that K is minimal, that is, that no proper subset of K is a key. However, this minimality assumption is not useful at the instance level. Thus we wish a relation in which a proper subset of K happens to uniquely identify tuples to be a possible instance of a schema in which K is a key. So our definition, in which minimality is not assumed, is more convenient for our purposes. («A Normal Form for Relational Databases That Is Based on Domains and Keys», Ronald Fagin)

Графическое пояснение ключевой зависимости (см. рисунок 3.2.t) выглядит не менее тривиально. Если допустить, что атрибут `e_passport` является ключом отношения `employee`, то ключевая зависимость присутствует в первом случае (значений-дубликатов в данном поле нет) и отсутствует во втором случае (значения-дубликаты в этом поле есть).

Ключевая зависимость присутствует

employee

...	e_passport	e_salary	...
...	AA123456	10 000	...
...	BB654321	12 000	...
...	CC112233	11 000	...
...	DD332211	10 500	...

Ключевая зависимость отсутствует

employee

...	e_passport	e_salary	...
...	**AA123456**	10 000	...
...	**AA123456**	10 000	...
...	CC112233	11 000	...
...	DD332211	10 500	...

*Рисунок 3.2.t — **Графическое пояснение ключевой зависимости***

Говоря о доменной зависимости, снова стоит уточнить, что фокус нашего внимания должен быть не на конкретной реализации (большинство СУБД предоставляет очень гибкие механизмы ограничения диапазонов и наборов значений поля), а на требованиях предметной области.

Так, эта зависимость присутствует в отношении (см. рисунок 3.2.u) тогда и только тогда, когда все значения некоего атрибута отношения являются представителями соответствующего домена{22} (предположим, что в отношении `event` атрибут `e_day_of_week` может принимать одно из семи значений: ПН, ВТ, СР, ЧТ, ПТ, СБ, ВС).

*Рисунок 3.2.u — **Графическое пояснение доменной зависимости***

Итак, теперь вы можете временно переключиться на изучение доменно-ключевой нормальной формы{273}.

Зависимости, на которых базируется шестая нормальная форма

Рассмотренная ранее зависимость соединения{196} актуальна для баз данных, не содержащих т.н. «хронологических данных[144]» (т.е. информации о моментах и отрезках времени и связанных с ними значениях некоторых атрибутов отношения).

Если же база данных предназначена для хранения и обработки таких данных, для неё становятся актуальными обобщённые реляционные операции[145] и обобщённые зависимости, в частности — обобщённая зависимость соединения.

Обобщённая зависимость соединения (generalized join dependency, U_join dependency[146]) — зависимость в отношении R (заголовок H которого содержит набор ACL атрибутов интервального типа), обозначаемая USING(ACL): ¤ {X_1, X_2, ..., X_n} и порождающая следующее ограничение: объединение (согласно теории множеств) проекций X_1, X_2, ..., X_n должно порождать отношение с тем же самым заголовком H.

*Упрощённо: взаимная зависимость между несколькими (более двух) атрибутами отношения (включая **интервальные**), обязывающая отношение содержать только такие данные, при которых декомпозиция отношения на проекции из групп этих атрибутов будет декомпозицией без потерь (т.е. позволяющей восстановить в точности исходное отношение).*

Фактически, в упрощённой формулировке этой зависимости было добавлено лишь уточнение «включая интервальные», в остальном же она эквивалентна упрощённой формулировке «обычной» зависимости соединения{196}.

Техническое примечание: в своей книге[147] К. Дж. Дейт поясняет, почему эта зависимость завязана именно на заголовок отношения, а не на само отношение, но для некоторого упрощения нашего материала оставим этот вопрос на самостоятельную проработку первоисточника.

По аналогии с соответствующими формами «обычной» зависимостью соединения сформулируем определения тривиальной и нетривиальной **обобщённой** зависимости соединения:

Тривиальная обобщённая зависимость соединения (trivial U_join dependency[148]) — такая зависимость соединения USING(ACL): ¤ {X_1, X_2, ..., X_n}, в которой хотя бы один компонент X_i содержит в себе полный набор атрибутов отношения R (т.е. весь его заголовок H).

Упрощённо: обобщённая зависимость соединения, которая выполняется всегда (в силу того факта, что отношение всегда можно в точности восстановить «из самого себя» (именно этот эффект достигается, если хотя бы одна из проекций содержит полный набор атрибутов отношения, т.е. является, фактически, самим же исходным отношением)).

---

144 См. главу 23 («Temporal Databases») в книге «An Introduction to Database Systems (8th edition)» (C.J. Date), где автор посвящает этому вопросу около 50 страниц с очень подробным описанием и примерами.

145 См. всё ту же главу 23 («Temporal Databases») в книге «An Introduction to Database Systems (8th edition)» (C.J. Date). Поскольку хронологические («темпоральные») базы данных частично выходят за рамки нашей тематики, мы ограничимся лишь кратком рассмотрением ключевых определений.

146 Let H be a heading, and let ACL be a commalist of attribute names in which every attribute mentioned (a) is one of the attributes in H and (b) is of some interval type. Then a U_join dependency (U_JD) with respect to ACL and H is an expression of the form USING (ACL) : ¤ {X_1,X_2,...,X_n}, such that the set theory union of X_1,X_2,...,X_n is equal to H. («The New Relational Database Dictionary», C.J. Date)

147 «The New Relational Database Dictionary», C.J. Date, стр. 423.

148 Определение сформулировано по аналогии с определением тривиальной зависимости соединения{194}.

Нетривиальная обобщённая зависимость соединения (nontrivial U_join dependency[149]) — такая зависимость соединения USING(ACL): ¤ {X_1, X_2, ..., X_n}, в которой ни один компонент X_i не содержит в себе полный набор атрибутов отношения R.

Упрощённо: обобщённая зависимость соединения, которая может не выполняться, т.е. может существовать ситуация, при которой отношение нельзя без искажений восстановить из его проекций на входящие в зависимость группы интервальных атрибутов.

И перед тем, как рассматривать пример, приведём ещё два определения, актуальных для хронологических баз данных.

Важно! Вне контекста хронологических баз данных следующие два термина могут иметь иную трактовку.

Г**оризонтальная декомпозиция** (horizontal decomposition[150]) — декомпозиция отношения, содержащего хронологические данные, на отношения, содержащие явные интервалы («с… по…») и содержащие неявные интервалы («с… по текущий момент», «с текущего момента по…»).

Упрощённо: информация об интервалах вида «с… по…» (т.е. имеющих начальный и конечный момент во времени) помещается в одни отношения, а информация об интервалах вида «с… по текущий момент» и «с текущего момента по…» (т.е. имеющих только один фиксированный момент во времени — начальный или конечный) помещается в другие отношения.

Вертикальная декомпозиция (vertical decomposition[151]) — декомпозиция хронологического интервального отношения, не находящегося в 6НФ{277}, в набор хронологических интервальных отношений, находящихся в 6НФ{277}.

Упрощённо: исходный набор атрибутов хронологического отношения, не находящегося в 6НФ{277}, перерабатывается таким образом, чтобы получить новые наборы атрибутов (возможно, не совпадающих с исходными) новых хронологических отношений, находящихся в 6НФ{277}. Ситуация, представленная на рисунке 3.2.v является наглядным примером вертикальной декомпозиции.

150151

Теперь переходим к примеру (см. рисунок 3.2.v). В исходном отношении `education_path` присутствует нетривиальная обобщённая зависимость соединения, что наглядно показано созданием двух проекций `P1` и `P2` этого отношения, позволяющих с помощью операции обобщённого соединения (`U_join`[152]) получить в точности исходное отношение.

---

149 Определение сформулировано по аналогии с определением нетривиальной зависимости соединения{194}.

150 **Horizontal decomposition** — informal term used to refer to the decomposition of a temporal relvar into a combination of *since* relvars and *during* relvars. («The New Relational Database Dictionary», C.J. Date)

151 **Vertical decomposition** — informal term used to refer to the decomposition (via U_projection) of a *during* relvar that's not in sixth normal form into a set of *during* relvars that are. («The New Relational Database Dictionary», C.J. Date)

152 См. раздел 23.5 («Generalizing the relational operators») в книге «An Introduction to Database Systems (8th edition)» (C.J. Date).

Обратите внимание: в отличие от «обычных» баз данных и связанных с ними операций декомпозиции, здесь значение атрибута `ep_period` в проекциях P_1 и P_2 получены не копированием значений из исходного отношения, а вычислением новых значений (в данном случае — на основе суммирования временных интервалов).

Исходное отношение

education_path

PK			
ep_student	**ep_period**	**ep_faculty**	**ep_group**
13452	01.01.2018-31.05.2018	Химический	1
13452	01.03.2018-31.05.2018	Химический	5
13452	01.06.2018-01.12.2018	Физический	5

Проекции

P_1 (ep_student, ep_period, ep_faculty)

PK		
ep_student	**ep_period**	**ep_faculty**
13452	01.01.2018-31.05.2018	Химический
13452	01.06.2018-01.12.2018	Физический

P_2 (ep_student, ep_period, ep_group)

PK		
ep_student	**ep_period**	**ep_group**
13452	01.01.2018-31.05.2018	1
13452	01.03.2018-01.12.2018	5

*Рисунок 3.2.v — **Пример обобщённой зависимости соединения***

Проекции P_1 и P_2 не могут быть вертикально декомпозированы далее (одна из получаемых проекций будет в точности совпадать с исходным отношением), таким образом получается, что в проекциях P_1 и P_2 присутствует тривиальная обобщённая зависимость соединения (или, если так проще запомнить: отсутствует нетривиальная обобщённая зависимость соединения).

И в заключение рассмотрим небольшой пример, поясняющий суть горизонтальной декомпозиции (ранее этот термин не был задействован в рассуждениях об обобщённой зависимости соединения, однако он достаточно важен).

Исходные отношения до горизонтальной декомпозиции

P_1

PK		
ep_student	**ep_period**	**ep_faculty**
13452	01.01.2018-31.05.2018	Химический
13452	01.06.2018-01.12.2018	Физический
13452	02.12.2018-СЕЙЧАС	Биологический

P_2

PK		
ep_student	**ep_period**	**ep_group**
13452	01.01.2018-31.05.2018	1
13452	01.03.2018-01.12.2018	5
13452	02.12.2018-СЕЙЧАС	10

Новые отношения после горизонтальной декомпозиции

P_{1_DURING}

PK		
ep_student	**ep_period**	**ep_faculty**
13452	01.01.2018-31.05.2018	Химический
13452	01.06.2018-01.12.2018	Физический

P_{1_NOW}

PK		
ep_student	**ep_start**	**ep_faculty**
13452	02.12.2018	Биологический

P_{2_DURING}

PK		
ep_student	**ep_period**	**ep_group**
13452	01.01.2018-31.05.2018	1
13452	01.03.2018-01.12.2018	5

P_{2_NOW}

PK		
ep_student	**ep_start**	**ep_group**
13452	02.12.2018	10

*Рисунок 3.2.w — **Пример горизонтальной декомпозиции***

Представим, что в отношения **P1** и **P2** необходимо добавить информацию о том, что с 02.12.2018 по настоящий момент времени студент с идентификатором 13452 обучается на биологическом факультете в группе 10. На рисунке 3.2.w показаны сразу обе ситуации — до горизонтальной декомпозиции и после неё.

Почему интервалы вида «с… по…» и вида «с… по текущий момент» не стоит хранить в одних и тех же отношениях? К. Дж. Дейт даёт достаточно подробное пояснение[153], но если вкратце: возникает огромный набор вопросов относительно даже самых простых операций (сравнение, сложение, вычитание) для понятия «сейчас»; для случаев, когда «сейчас» является началом интервала, возникает необходимость дополнительных операций управления данными (самый простой пример — рано или поздно «сейчас» станет больше, чем «конец интервала»). Одним словом — выгода минимальна, а потенциальных проблем — уйма.

Да, в любой современной СУБД есть возможность получить текущее значение даты и времени и сравнить полученное с любым датавременным значением. Да, многие СУБД позволяют автоматически обновлять значение датавременного типа при обращении к записи в таблице. Да, можно использовать значение «сейчас» при построении практически любого SQL-запроса. Всё это так, и тем не менее это вовсе не отменяет обозначенных К. Дж. Дейтом проблем (большинство из которых имеет лишь ограниченное решение, а некоторые не решены вовсе).

Вывод прост: храните информацию о закрытых интервалах (вида «с… по…») и об открытых интервалах (вида «с… по текущий момент» и «с текущего момента по…») в разных отношениях.

Итак, теперь вы можете временно переключиться на изучение шестой нормальной формы{277}.

Итоговый обзор всех ключевых зависимостей

Только что мы рассмотрели лишь самые ключевые разновидности зависимостей, изучение которых совершенно необходимо для понимания нормальных форм.

В контексте реляционной теории принято рассматривать куда больше зависимостей (и их свойств), используемых в формальных доказательствах алгоритмов нормализации. Если этот материал представляет для вас интерес, обратитесь к следующей литературе:

- «Fundamentals of Database Systems, 6th edition» (Ramez Elmasri, Shamkant Navathe) — главы 15-16.
- «An Introduction to Database Systems, 8th edition» (C.J. Date) — главы 11-13.

Мы же для закрепления материала ещё раз приведём список только что рассмотренных зависимостей и приведём их определения в ещё более сжатой и упрощённой форме, чем это было сделано ранее.

Аналогичный краткий список всех нормальных форм будет представлен далее{280}.

Пожалуйста, используйте приведённые ниже определения только как способ быстрого запоминания сути той или иной зависимости. Полноценные строгие определения можно быстро найти, проследовав по ссылке, представленной рядом с англоязычным вариантом каждого термина.

---

153 См. раздел 23.6 («Database design», подзаголовок «The moving point now») в книге «An Introduction to Database Systems (8th edition)» (C.J. Date).

Функциональная зависимость (functional dependency{180}) — зависимость, в которой одному значению детерминанта соответствует одно значение зависимой части.

Полная функциональная зависимость (full functional dependency{183}) — зависимость, в которой значение зависимой части зависит от всех частей детерминанта.

Частичная функциональная зависимость (partial functional dependency{184}) — зависимость, в которой значение зависимой части зависит лишь от некоторых частей детерминанта.

Тривиальная функциональная зависимость (trivial functional dependency{191}) — функциональная зависимость, которая не может быть нарушена (в силу того факта, что зависимая часть является частью детерминанта).

Нетривиальная функциональная зависимость (nontrivial functional dependency{191}) — функциональная зависимость, которая может быть нарушена (в силу того факта, что зависимая часть не является частью детерминанта).

Транзитивная зависимость (transitive dependency{187}) — «цепочка» зависимостей (две и более функциональных зависимости, в которых зависимая часть предыдущей является детерминантом следующей).

Избыточная зависимость (redundant dependency{187}) — зависимость, которая может быть вычислена на основе других зависимостей.

Избыточная транзитивная зависимость (redundant transitive dependency{187}) — «цепочка» зависимостей, в которых существует функциональная зависимость между детерминантом первой и зависимой частью последней зависимостей.

Избыточная псевдотранзитивная зависимость (pseudotransitive dependency{187}) — «цепочка» зависимостей, в которых детерминант последней зависимости можно вычислить на основе зависимых частей нескольких предыдущих зависимостей.

Многозначная зависимость (multivalued dependency{193}) — зависимость, которая обязывает отношение, содержащее два кортежа, совпадающие по значению одного из трёх атрибутов, содержать также два дополнительных кортежа, «перекрёстно» содержащие комбинации оставшихся двух атрибутов.

Тривиальная многозначная зависимость (trivial multivalued dependency{194}) — многозначная зависимость, в которой для группы из трёх атрибутов X, Y, Z либо Y входит в состав X, либо Z отсутствует.

Нетривиальная многозначная зависимость (nontrivial multivalued dependency{194}) — многозначная зависимость, в которой для группы из трёх атрибутов X, Y, Z атрибут Y не входит в состав X, и атрибут Z присутствует.

Зависимость соединения (join dependency{196}) — зависимость, которая обязывает отношение явным образом содержать все кортежи, которые получатся после восстановления отношения из его проекций на группы атрибутов, входящих в состав этой зависимости.

Тривиальная зависимость соединения (trivial join dependency{196}) — зависимость соединения, в которой хотя бы одно множество атрибутов, составляющих набор проекций, на которых построена данная зависимость, представляет собой полный набор атрибутов отношения.

Нетривиальная зависимость соединения (nontrivial join dependency{196}) — зависимость соединения, в которой ни одно из множеств атрибутов, составляющих набор проекций, на которых построена данная зависимость, не представляет собой полный набор атрибутов отношения.

Доменная зависимость (domain dependency{201}) — зависимость, которая обязывает все значения любого атрибута отношения принадлежать множеству допустимых значений для данного атрибута.

Ключевая зависимость (key dependency{201}) — зависимость, которая обязывает все значения ключа отношения быть уникальными.

Обобщённая зависимость соединения (generalized join dependency{204}) — зависимость, которая обязывает хронологическое отношение явным образом содержать все атрибуты и кортежи, которые получатся после восстановления отношения из его проекций на группы атрибутов, входящих в состав этой зависимости и содержащих хотя бы один интервальный атрибут.

Тривиальная обобщённая зависимость соединения (trivial U_join dependency{204}) — зависимость соединения, в которой хотя бы одно множество атрибутов, составляющих набор проекций, на которых построена данная зависимость, представляет собой полный набор атрибутов хронологического отношения.

Нетривиальная обобщённая зависимость соединения (nontrivial U_join dependency{205}) — зависимость соединения, в которой ни одно из множеств атрибутов, составляющих набор проекций, на которых построена данная зависимость, не представляет собой полный набор атрибутов хронологического отношения.

Итак, мы уже почти готовы рассматривать нормальные формы. Осталось лишь привести несколько важных рассуждений о процессе их применения, чем и будет посвящена следующая глава.

Задание 3.2.a: запишите (в любой удобной вам нотации) все функциональные зависимости, присутствующие в базе данных «Банк»{408}.

Задание 3.2.b: существуют ли в базе данных «Банк»{408} отношения, содержащие многозначные зависимости? Если вы считаете, что «да», доработайте модель так, чтобы устранить такие зависимости.

Задание 3.2.c: составьте список зависимостей, рассмотренных в данной главе, но отсутствующих в базе данных «Банк»{408}.

3.2.2. ТРЕБОВАНИЯ НОРМАЛИЗАЦИИ В КОНТЕКСТЕ ПРОЕКТИРОВАНИЯ

Начнём с главного определения.

Нормализация (normalization[154]) — процесс декомпозиции переменной отношения R на набор проекций R_1, R_2, ..., R_n, таких, что:
а) объединение проекций R_1, R_2, ..., R_n позволяет гарантированно получить исходную переменную отношения R;
б) каждая из проекций R_1, R_2, ..., R_n является необходимой для выполнения условия «а»;
в) как минимум одна из проекций R_1, R_2, ..., R_n находится в более высокой нормальной форме, чем исходная переменная отношения R.

Упрощённо: переменная отношения разбивается на несколько новых, находящихся в более высоких нормальных формах и позволяющих получить исходную переменную отношения использованием объединения (JOIN).

Прежде, чем приступить к разговору о процессе нормализации, стоит рассмотреть, к чему нужно стремиться, на что нужно ориентироваться и чего стоит избегать, выполняя нормализацию — это всё может быть сведено к единому перечню требований нормализации[155], которые не являются строгим и исчерпывающим списком, но всё же дают достаточно ориентиров.

Процесс нормализации является неотъемлемой частью проектирования баз данных, а потому позволяет выявить и исправить многие недостатки проектируемой схемы — как напрямую связанные с нарушением нормальных форм, так и многие иные, но не менее опасные.

Поскольку нормализация невозможна без переработки схемы базы данных, соответствующие действия обязательно затронут проектную документацию, и потому часть далее рассмотренных требований относится к документированию процесса проектирования.

Также отметим, что представленные далее требования в некоторых случаях могут противоречить друг другу, и это совершенно нормально — нет задачи формально добиться соблюдения их всех, есть лишь необходимость учитывать те требования, которые в наибольшей мере соответствуют ключевым показателям качества разрабатываемой базы данных.

Однако, для начала рассмотрим основную идею, выраженную несколькими принципами нормализации (normalization principles[156]):

- Переменная отношения, не находящаяся в нормальной форме неотъемлемых кортежей (НФНК, ETNF{280}) должна быть декомпозирована в набор проекций, находящихся в НФНК или даже более высоких нормальных формах.

---

154 **Normalization** is a replacing a relvar R by certain of its projections R_1, R_2, ..., R_n, such that (a) the join of R_1, R_2, ..., R_n is guaranteed to be equal to R, and usually also such that (b) each of R_1, R_2, ..., R_n is needed in order to provide that guarantee (i.e., none of those projections is redundant in the join), and usually also such that (c) at least one of R_1, R_2, ..., R_n is at a higher level of normalization than R is. The usual objective of normalization is to reduce redundancy and thereby to eliminate certain update anomalies that might otherwise occur. («The New Relational Database Dictionary», C.J. Date)

155 В классических первоисточниках эти требования фрагментарно представлены в ранних работах Кодда и Дейта, а в виде единого краткого списка впервые сформулированы в курсе «Основы проектирования реляционных баз данных» [http://www.intuit.ru/studies/courses/1095/191/info], после чего многократно скопированы во множество русскоязычных материалов. Проблема в том, что краткое перечисление оказывается недостаточным для понимания начинающими разработчиками баз данных, потому здесь такие пояснения приведены, а сам список переработан.

156 **Normalization principles** is a set of principles used to guide the practical process of normalization. The principles in question are as follows: (a) A relvar not in ETNF should be decomposed into a set of projections that are in ETNF (and possibly some higher normal form, such as 5NF or 6NF); (b) the original relvar should be reconstructable by joining those projections back together again; (c) the decomposition process should preserve dependencies; (d) every projection should be needed in the reconstruction process. («The New Relational Database Dictionary», C.J. Date)

- Исходная переменная отношения должна быть восстановима через объединение получившихся проекций.
- Декомпозиция исходной переменной отношения в проекции должна сохранять зависимости, имевшиеся в исходной переменной отношения.
- В процессе декомпозиции каждая полученная проекция должна быть необходимой для восстановления исходной переменной отношения.

Несмотря на то, что в этом перечне речь идёт про т.н. «неканоническую» нормальную форму неотъемлемых кортежей, с научной точки зрения тут всё правильно. С практической же точки зрения очень редко переменная отношения, находящаяся в 4НФ{263} не будет находиться в НФНК{280}, потому в процессе нормализации ориентируются именно на 4НФ{263} (а иногда и на 3НФ{252} или НФБК{258}, если такой степени нормализации оказывается достаточно для устранения всех имеющихся аномалий операций с данными{161}).

И теперь перейдём к рассмотрению более объёмных, но не менее важных требований, которые стоит учитывать, выполняя нормализацию.

Сохранение ограничений предметной области

В процессе нормализации стоит особое внимание уделять результатам декомпозиции исходных переменных отношений на их проекции{176}. Такие результаты должны:

- удовлетворять требованию декомпозиции без потерь{180};
- сохранять явно или позволять восстанавливать при выполнении операций объединения все ранее существовавшие зависимости и ограничения.

Нарушение этого требования является крайне нежелательным, т.к. потребует создания дополнительных (как правило — весьма нетривиальных) механизмов обеспечения консистентности{72} базы данных.

Соблюдение общих требований к модели базы данных

Общие универсальные требования к любой базе данных{12} были рассмотрены в самом начале этой книги. Напомним их кратко: адекватность предметной области, удобство использования, производительность, защищённость данных. Эти требования настолько фундаментальны, что уделять им внимание стоит на любом уровне проектирования базы данных и при выполнении любых манипуляций с её схемой, т.к. при нарушении хотя бы части из этого набора требований мы рискуем получить базу данных, непригодную к дальнейшей эксплуатации.

Требование минимальности первичных ключей

Казалось бы, про свойство минимальности{39} первичных ключей{39} уже сказано столько, что можно было бы и не повторяться. Увы, на практике с этим требованием бывают проблемы.

В общем случае можно задать себе три вопроса относительно первичного ключа некоторой таблицы:

- Нужен ли таблице первичный ключ?
- Достаточно ли он мал?
- Достаточно ли он велик?

Нужен ли таблице первичный ключ? Интуитивно кажется очевидным, что первичный ключ нужен всегда, ведь иначе мы не сможем гарантированно различить строки таблицы, не сможем устанавливать связи и получим ещё уйму негативных последствий. Это так, но — не всегда. Бывают исключения, когда наличие первичного ключа не только не является необходимым, но напротив — сильно мешает. Пример такого случая будет рассмотрен очень скоро{216}. Пока же согласимся, что чаще всего первичный ключ нужен, и вопрос лишь в том, что он из себя представляет.

Достаточно ли имеющийся первичный ключ мал? В первую очередь стоит сделать выбор между естественным{42} и искусственным{42} первичными ключами (что часто влечёт за собой выбор между составным{40} и простым{40} первичным ключом).

Не погружаясь в длинные философские споры, согласимся, что составной естественный первичный ключ тоже имеет право не существование, но в абсолютном большинстве случаев вполне справедливый выбор будет сделан в пользу простого искусственного, где для обеспечения минимальности останется лишь выбрать наиболее подходящий тип данных.

Если по каким-то причинам целочисленный тип нам не подходит, обязательно стоит изучить, как конкретная СУБД хранит и обрабатывает выбранный вами тип данных. Иногда можно получить крайне неприятные сюрпризы в виде «выравнивания» строк в кодировке UTF8 по границе «четыре байта на символ», дополнения хранимых данных заданными величинами (как правило, пробелами или нулями) до некоторой фиксированной длины и т.д. Здесь поможет только вдумчивое изучение документации.

Но представим хорошую ситуацию: целочисленный тип нам подходит, и мы выбрали даже один из самых «коротких» типов — для простоты иллюстрации предположим, что мы работаем с MySQL и выбрали тип данных `TINYINT` (фактически — байт).

Достаточно ли имеющийся первичный ключ велик? Это — самый простой вопрос из всех трёх, но о нём почему-то реже всего задумываются. Для получения верного ответа нужно понимать, какое количество записей будет храниться в таблице, и как будет происходить их изменение со временем.

В нашем примере мы выбрали тип данных, который может хранить 256 значений. Этого будет совершенно достаточно для присвоения уникальных идентификаторам всем записям в том случае, если их немного, и они со временем не добавляются (или добавляются лишь в исключительных случаях). Например, такого первичного ключа должно хватить для таблиц с перечнем стран, статусов пользователей, форм проведения платежей и т.д.

Но такого типа данных совершенно точно не хватит для хранения списка сотрудников, списка статей, списка банковских транзакций и т.д. Ситуацию усугубляет тот факт, что подобный первичный ключ часто делают автоинкрементируемым, а потому даже удаление из таблицы устаревших записей не высвобождает часть ранее использованных значений такого первичного ключа[157] — оно постепенно и неотвратимо приближается к максимальной границе, по достижению которой вставка записей станет невозможной.

Потому здесь есть универсальный совет: не стоит «экономить на спичках». Пара выигранных байтов на одну запись даст вам в масштабах типичной базы данных экономию доли процента объёма и прирост производительности на неуловимо малую величину, но потенциально приведёт к отказу в самый неподходящий момент. Это — ни в коем случае не призыв делать первичные ключи огромными. Просто оставьте некоторый *разумный* запас.

---

[157] На самом деле, решение есть, но оно совсем не тривиальное. См. пример 38 в книге «Работа с MySQL, MS SQL Server и Oracle в примерах» (С.С. Куликов) [http://svyatoslav.biz/database_book/]

Требование неизбыточности данных

Ранее рассмотренные аномалии обновления{162} и удаления{162} процветают в случае, когда одни и те же данные хранятся в нескольких строках одной таблицы. Ситуация оказывается весьма схожей, когда одни и те же данные хранятся в разных таблицах.

Представим, что в некоторой базе данных сложилась следующая ситуация (см. рисунок 3.2.x): в двух отношениях (`contract` и `finance`) хранится информация о договорах и соответствующих им суммах. И мы видим, что для одного договора суммы не совпадают. При условии, что по требованиям предметной области это одна и та же сумма (т.е. числа обязаны быть одинаковыми) — каким данным верить? Хорошо, если где-то есть дополнительные источники информации, по которым можно выяснить истину (при условии, что мы заметили ошибку, а ведь могли и не заметить) — а если таких источников нет? Всё, тупик: одни данные точно неверны, и мы никак не можем выяснить, какие именно.

Такая ситуация была бы исключена, если бы сумма хранилась строго в одном месте.

Иногда можно услышать возражение, что, сохраняя данные в нескольких местах, мы повышаем шансы не потерять их в случае сбоев и повреждения хранилища. Это в корне неверный подход: ни в коем случае не стоит путать безусловно важную задачу создания резервных копий, зеркалирования и иных мер по обеспечению сохранности данных и задачу проектирования базы данных. Нельзя инструментами решения одной из этих задач решать другую.

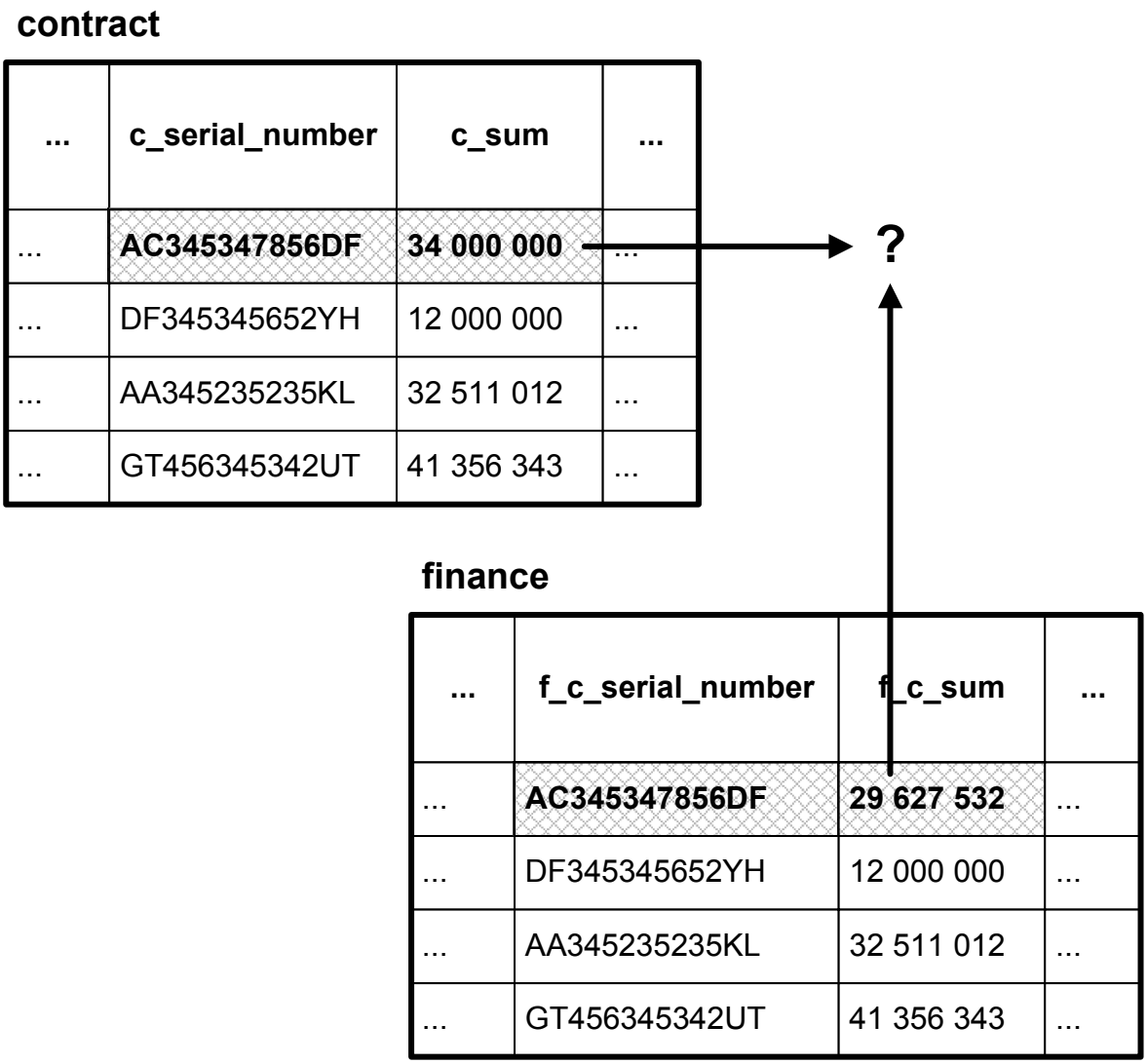


*Рисунок 3.2.x — **Нарушение требования неизбыточности данных***

Требование производительности системы

О производительности базы данных немного было сказано ранее{16}, но сейчас мы подойдём к этому вопросу с другой стороны: какими характеристиками модели базы данных, влияющими на производительность, мы можем легко управлять на стадии *проектирования*? Особо подчеркнём: на стадии проектирования, т.е. здесь мы пока не затрагиваем настройки СУБД, параметры аппаратного обеспечения, сетевую инфраструктуру и т.д.

Мы можем вспомнить про минимальность первичных ключей{212} (и индексов, т.к. всё, сказанное о ключах, в полной мере относится и к индексам), взвесить необходимость создания тех или иных индексов, продумать необходимую глубину нормализации[158] схемы и возможность использования денормализации{233} как одного из способов повышения производительности.

Рассмотрим простой пример оптимизации в контексте производительности схемы базы данных, состоящей из трёх таблиц (см. рисунок 3.2.y) — допустим, что перед нами база данных некоего вымышленного новостного сайта.

Предположим, что на основе исследования аналогичных продуктов было выяснено следующее:

[158] Это не строгий термин, но в общем случае он означает ту максимальную нормальную форму, до которой мы собираемся нормализовать каждую отдельно взятую переменную отношения.

- к таблице **`news`** в день в среднем происходит 200-300 тысяч обращений на чтение, 5-10 на запись и 2-3 на обновление/удаление;
- к таблице **`log`** в день происходит 500-700 тысяч обращений на запись, никогда не происходит обращений на обновление/удаление, а обращения на чтение происходят 1—2 раза в неделю, но их результаты нужно получать очень быстро.

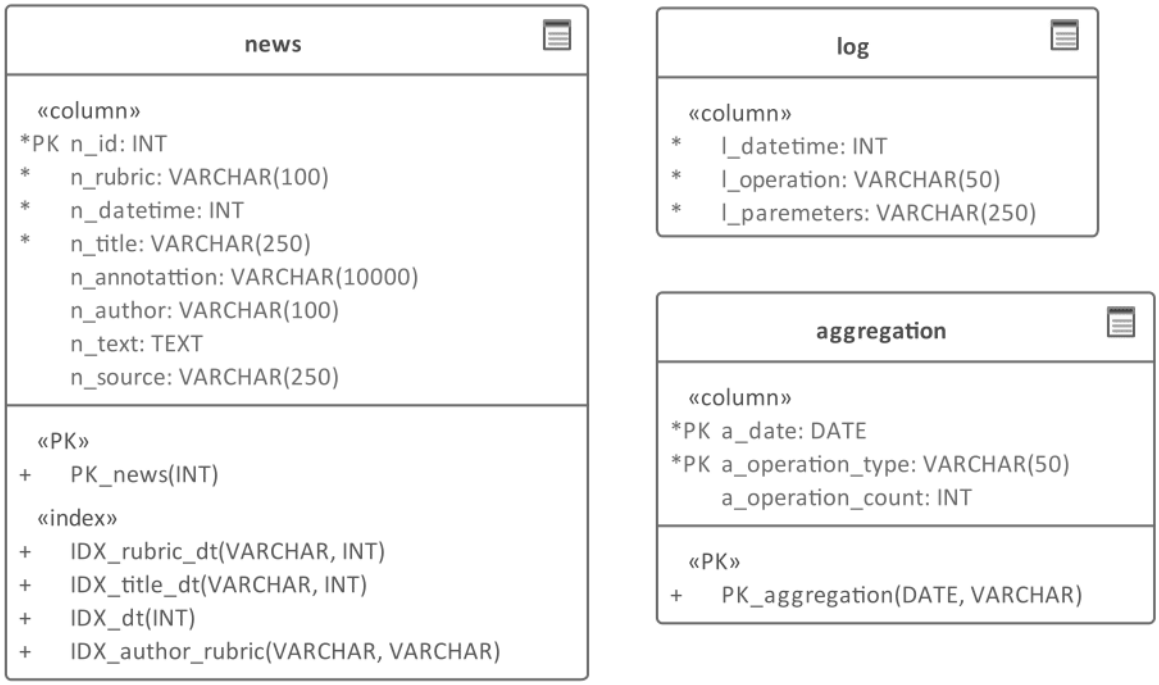


*Рисунок 3.2.y — **Обеспечение максимальной производительности базы данных на стадии проектирования***

Таблица **`news`** является ярким примером случая «оптимизации на чтение», т.е. мы можем изначально создать несколько индексов, не переживая, что их наличие затормозит выполнение операций модификации данных.

Таблица **`log`** является ярким примером случая «оптимизации на запись», и потому мы полностью убрали с неё какие бы то ни было индексы, включая первичный ключ (именно данный случай является обещанным ранее{212} примером ситуации, когда таблице первичный ключ не нужен). В результате мы пожертвовали скоростью чтения, сильно выиграв в скорости записи.

Но у нас осталась одна нерешённая проблема: сказано, что чтение данных из таблицы **`log`** должно происходить очень быстро, а мы только что поставили крест на быстром чтении из этой таблицы. Но решение всё равно есть: необходимо выяснить, какие операции чтения будут выполняться и насколько актуальные будут необходимы данные, а затем подготавливать результаты заранее.

Допустим, мы выяснили, что заказчику необходимо знать, сколько раз в сутки выполнялась та или иная операция с сайтом, а актуальность информации его устраивает «в пределах суток». Тогда мы создадим таблицу **`aggregation`**, которая будет содержать результаты соответствующей выборки, а саму выборку будем проводить раз в сутки в наименее нагруженное для сайта время (например, ночью).

Да, каждую конкретную таблицу мы оптимизировали для выполнения лишь одного вида операций (либо чтение, либо модификация данных), но в целом такая схема максимально быстро реагирует на все наиболее характерные для неё операции с данными, чего мы и хотели добиться.

Легко заметить, что таблица `news` на рисунке 3.2.y подвержена ряду аномалий операций с данными{161}, которые можно (и нужно будет!) предотвратить с помощью нормализации.

Требование непротиворечивости данных

Это требование — одновременно одно из самых простых, и одно из самых сложных.

Простота заключается в том, что, фактически, здесь речь идёт о консистентности{72} данных и о том, что схема базы данных не должна создавать предпосылок для её нарушения.

Сложность же разделяется на два направления.

Во-первых, сложно соблюсти все ограничения консистентности — их очень много, и высока вероятность о некоторых из них не узнать вообще (т.е. допустить просчёт на стадии анализа предметной области), какие-то забыть реализовать с помощью соответствующих механизмов БД/СУБД, а какие-то реализовать некорректно (или забыть обновить в соответствии с изменениями схемы базы данных или требованиями предметной области). Подробнее о том, как решается эта проблема, сказано в соответствующем разделе{284}.

Во-вторых, очень велик соблазн переложить реализацию некоторых ограничений на уровень работающих с базой данных приложений или вовсе отказаться от реализации части ограничений, исходя из предположения, что «и так всё будет нормально работать».

Во многом именно по данной причине такое фундаментальное требование к любым реляционным базам данных повторяется многократно в самых различных контекстах: подавляющее большинство катастрофических проблем с базами данных происходит именно потому, что кто-то когда-то посчитал ту или иную фатальную ситуацию маловероятной или вовсе невозможной. Но она возникла, нанеся непоправимый ущерб. Вывод предельно прост: если вы понимаете, что некое ограничение необходимо, его стоит реализовать средствами БД/СУБД, не надеясь, что кто-то где-то сделает это за вас.

И ещё одно проявление рассматриваемой сложности достойно отдельного упоминания: полная реализация всех необходимых ограничений может вступать в противоречие с требованиями производительности{215} и гибкости структуры{218} базы данных.

К сожалению, универсального решения здесь нет.

При наличии достаточного количества ресурсов, безусловно, стоит отдать предпочтение требованию непротиворечивости, а вопросы производительности и гибкости структуры решить, соответственно, с помощью добавления аппаратных ресурсов и тщательного документирования (также помня о том, что в будущем переработка такой схемы потребует больше усилий).

На практике же приходится искать компромиссные решения, многие из которых могут выглядеть очень сомнительно с точки зрения теории баз данных, но при этом эффективно решать поставленную бизнес-задачу.

Требование гибкости структуры базы данных

При проектировании базы данных следует понимать, что создаваемая схема не является конечной, что она будет эволюционировать по мере развития проекта, и что изменения могут произойти как уже сегодня, так и через большой промежуток времени.

Данное требование нормализации призывает помнить о грядущих изменениях, и готовиться к ним уже сейчас.

Только что{217} мы увидели, что у этого требования есть «естественный враг» в виде огромного количества разнообразных ограничений, объективно необходимых для обеспечения консистентности{72} данных, и что эта проблема требует частных решений в контексте поставленной бизнес-задачи.

Но существует также целый ряд других проблем, способных превратить эволюционное развитие базы данных в ад. Эти проблемы тривиальны, имеют простые и понятные решения, но, к сожалению, сплошь и рядом встречаются в работах начинающих специалистов по базам данных. Все они сводятся к нарушению одного или нескольких представленных далее правил.

Имена всех структур должны быть мнемоничными и следовать единому соглашению об именовании. Частично мы уже затрагивали это правило{27} в разделе, посвящённом отношениям. Вкратце вся суть сводится к следующему: используйте единообразно оформленные имена, значения которых позволяют понять суть и назначение именованной структуры.

Только что{217} была подчёркнута сложность продумывания всех необходимых ограничений консистентности, а при проблеме с именованием структур базы данных эта задача становится практически нерешаемой — вместо обдумывания некоего сложного правила проектировщик тратит все силы в попытках просто понять, что его предшественник (или даже он сам) имел в виду.

Сравните:

	Плохо	Хорошо
Имена таблиц	data	registered_user
	connection	m2m_user_role
	t1	news_rubric
Имена полей	pp	u_primary_phone
	ok	o_is_order_confirmed
	some_text_data	a_biography
Имена хранимых функций	process	get_initials
	get_date	unixtime_to_datetime
	fnc	upcase_first_letters
Имена триггеров	no_admins	t_user_protect_last_admin_del
	update_all	t_aggregate_user_stats_upd
	bad_dates	t_date_start_le_end_ins_upd
Имена индексов	qsrch	idx_rubric_author_date
	singlerecord	unq_login
	tr	idx_author_btree

Фактически, к именам структур баз данных можно применять большинство правил именования из классического программирования. Да, в некоторых СУБД существуют достаточно строгие ограничения, не позволяющие создать очень длинное и подробное имя, но даже в такой ситуации можно сохранить основную суть, заменив аббревиатурами лишь универсальные и/или не самые важные части имени.

Отдельно хочется сказать несколько слов об использовании префиксов в именах: именовать ли индексы с префикса `idx` (а уникальные индексы — с префикса `unq`), начинать ли имена полей таблицы с аббревиатуры из имени самой таблицы и т.д. Это очень «религиозный» вопрос, в котором каждый из вариантов ответов имеет множество сторонников и противников.

Выбор за вами. Главное — сохранять последовательность в сделанном выборе, т.е., например, если вы решили начинать имена полей таблиц с аббревиатуры из имени таблицы, это правило должно выполняться для всех полей всех таблиц. Без исключения.

Схема базы данных должна быть снабжена комментариями. Это правило тоже полностью согласуется с классикой программирования. Любое современное средство проектирования позволяет писать комментарии к полям, таблицам, коду триггеров и хранимых подпрограмм и т.д. Причём такие комментарии сохраняются в самой базе данных и будут доступны даже в том случае, если сопроводительная документация по какой-то причине недоступна.

Сравните два фрагмента кода (в синтаксисе MySQL):

MySQL Код без комментариев

```
CREATE TABLE `file`
(
 `f_uid`              BIGINT UNSIGNED NOT NULL AUTO_INCREMENT,
 `f_fc_uid`           BIGINT UNSIGNED NULL,
 `f_size`             BIGINT UNSIGNED NOT NULL,
 `f_upload_datetime`  INTEGER NOT NULL,
 `f_save_datetime`    INTEGER NOT NULL,
 `f_src_name`         VARCHAR(255) NOT NULL,
 `f_src_ext`          VARCHAR(255) NULL,
 `f_name`             CHAR(200) NOT NULL,
 `f_sha1_checksum`    CHAR(40) NOT NULL,
 `f_ar_uid`           BIGINT UNSIGNED NOT NULL,
 `f_downloaded`       BIGINT UNSIGNED NULL DEFAULT 0,
 `f_al_uid`           BIGINT UNSIGNED NULL,
 `f_del_link_hash`    CHAR(200) NOT NULL
)
-- ...
```

MySQL Код с комментариями

```
CREATE TABLE `file`
(
 `f_uid`              BIGINT UNSIGNED NOT NULL AUTO_INCREMENT
                      COMMENT 'Глобальный идентификатор файла.',
 `f_fc_uid`           BIGINT UNSIGNED NULL
                      COMMENT 'Идентификатор категории файлов
                               (ВК на таблицу категорий).',
 `f_size`             BIGINT UNSIGNED NOT NULL
                      COMMENT 'Размер файла (в байтах).',
 `f_upload_datetime`  INTEGER NOT NULL
                      COMMENT 'Дата-время загрузки файла на
                               сервер (в Unixtime).',
 `f_save_datetime`    INTEGER NOT NULL
                      COMMENT 'Дата-время, до которых файл
                               хранится на сервере (в Unixtime).',
 `f_src_name`         VARCHAR(255) NOT NULL
                      COMMENT 'Исходное имя файла (как он назывался у
                               пользователя на компьютере). Без расширения,
                               т.к. оно хранится отдельно в поле
                               f_src_ext!',
```

```
  `f_src_ext`          VARCHAR(255) NULL
                       COMMENT 'Исходное расширение файла (каким оно
                                было у пользователя на компьютере).',
  `f_name`             CHAR(200) NOT NULL
                       COMMENT 'Имя файла на сервере. Пять SHA1-хешей.',
  `f_sha1_checksum`    CHAR(40) NOT NULL
                       COMMENT 'Контрольная сумма файла, SHA1-хеш.',
  `f_ar_uid`           BIGINT UNSIGNED NOT NULL
                       COMMENT 'Права доступа к файлу (ВК на таблицу
                                прав доступа).',
  `f_downloaded`       BIGINT UNSIGNED NULL DEFAULT 0
                       COMMENT 'Счётчик скачиваний файла.',
  `f_al_uid`           BIGINT UNSIGNED NULL
                       COMMENT 'Возрастные ограничения на доступ к файлу (ВК
                                на таблицу возрастных ограничений). Если
                                возрастные ограничения заданы и для файла,
                                и для категории, к которой принадлежит файл,
                                применяются более строгие ограничения (т.е.
                               берётся максимальный возраст, например: «с 16
                                лет» и «с 18 лет» — берётся «с 18 лет»).',
  `f_del_link_hash`    CHAR(200) NOT NULL
                       COMMENT 'Ссылка на удаление файла (пять SHA1-хешей),
                                если его размещал незарегистрированный
                                пользователь. Зарегистрированные могут
                                удалить файл в своём «личном кабинете».'
)
-- ...
```

Конечно, код без комментариев выглядит намного короче и компактнее — и это даже может быть преимуществом в том случае, если вы его написали пару дней назад и активно с ним работаете. Но если это не вами написанный код, или вы написали его месяцы (годы?) назад, вариант с комментариями оказывается в разы информативнее.

Процесс проектирования базы данных должен быть документирован. Это требование является логическим продолжением и расширением предыдущего: комментарии — это прекрасно, но одних лишь комментариев чаще всего оказывается недостаточно. Тем более, что комментарии будут относиться лишь к финальному состоянию схемы, а мы сейчас говорим именно о документировании всего процесса — от анализа предметной области до настроек СУБД и аппаратно-программного окружения.

Чем сложнее проектируемая база данных, чем больше противоречивых требований к ней предъявляется, чем больше нетривиальных нюансов в предметной области, тем большую пользу приносит полноценная документация, в которой не просто зафиксированы те или иные решения, но и даны пояснения: почему принято именно такое решение, какие были альтернативы, какие аргументы оказались решающими.

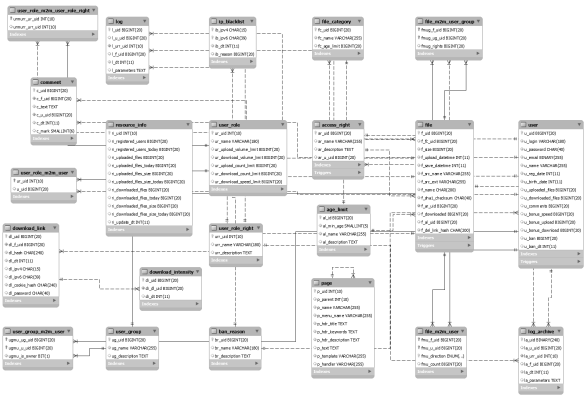
Если такой документации нет, «доработка» базы данных будет куда больше напоминать или хождение по минному полю, где любое неосторожное действие может привести к плачевным последствиям, или… проще будет разработать новую базу данных с нуля, чем безболезненно адаптировать к новым требованиям имеющуюся.

В документации должна быть представлена схема базы данных в общепринятой графической нотации. Это правило можно было бы объединить с предыдущим, но слишком часто звучат возражения, что графическое представление схемы можно получить в любой среде разработки баз данных через обратное проектирование{331}.

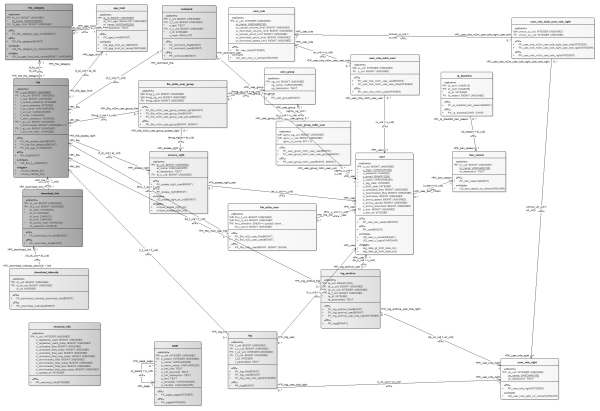
Можно. Но сравните два следующих случая (рисунок 3.2.z).

Здесь представлена микроскопическая по промышленным меркам база данных из двух десятков таблиц, но даже на таком объёме видно, что логически сгруппированные, отмеченные цветами и дополненные комментариями таблицы воспринимаются намного легче, чем просто нагромождение сваленных вперемешку объектов.

Когда речь идёт о базах данных из сотен таблиц, эта разница становится не просто существенной, а практически непреодолимой: попытки «очеловечить» автоматически сгенерированную схему могут занять дни и недели.



Полученная автоматически схема



Созданная вручную схема

*Рисунок 3.2.z — **Полученная автоматически и созданная вручную схемы одной и той же базы данных***

В схеме должны отсутствовать неаргументированные жёсткие ограничения и/или нетипичные решения. Пожалуй, ничто так не подрывает процесс доработки базы данных как наличие в ней «странных» и «нелогичных» решений, которые не отражены в документации или недостаточно объяснены.

В предельном случае наличие таких артефактов может означать одно из двух: или это решение было принято «от безысходности» как едва ли не единственный способ обеспечить выполнение некоего технологического или бизнес-правила, или… это просто чья-то глупость, т.е. решение, принятое бездумно.

Во втором случае, конечно же, стоит немедленно переработать схему, избавив её от очевидного недостатка. В первом случае такая переработка может привести к необратимым последствиям.

Потому любое нетипичное решение должно быть максимально подробно документировано и объяснено. Но ещё лучше — избегать подобных решений вовсе.

Требование актуальности данных

Как и многие другие, это требование может быть очень простым или очень сложным для выполнения, в зависимости от схемы данных, требований предметной области и множества других факторов.

В предельно простом случае его можно сформулировать так: избегайте создания кэширующих и агрегирующих таблиц[159], а если вы всё же вынуждены их создать, проработайте механизм своевременного обновления данных.

---

159 См. пример 31 в книге «Работа с MySQL, MS SQL Server и Oracle в примерах» (С.С. Куликов) [http://svyatoslav.biz/database_book/]

Ранее{216} на рисунке 3.2.y был приведён пример как раз такой ситуации: отношение **aggregation** является агрегирующим, т.е. хранит в себе обработанные (агрегированные) данные из отношения **log**. В этом примере мы договорились, что допускается использование агрегированных данных, устаревших по отношению к агрегируемым на сутки — это почти утопическая договорённость, использованная для простоты в учебном контексте. В реальности такое «допустимое отставание» может измеряться минутами, секундами, долями секунд.

Возникает не только проблема столь частого обновления агрегируемых данных, но и проблема гарантированного обеспечения их актуальности — не должно ни при каких обстоятельствах возникнуть ситуации, в которой приложение, работающее с нашей базой данных, получает устаревшие данные, будучи уверенным, что работает с актуальными.

Если не вдаваться в технические подробности[160] реализации механизмов инвалидации кэша, то самым частым решением (при условии, что оно технически реализуемо и приемлемо) будет обновление агрегированных/закэшированных данных так часто, чтобы «возраст» обновлённой копии всегда был не больше максимально допустимого «периода отставания».

Итак, становится понятно, что агрегирующие/кэширующие таблицы могут как принести пользу, так и создать сложности. А что, если попробовать обойтись без них?

Тогда мы переходим к более сложному случаю реализации рассматриваемого требования. С одной стороны, мы уже рискуем нарушить требование производительности{215}, т.к. выполнение некоторых запросов может занимать ощутимое время. С другой стороны, у нас всё равно остаётся вопрос: в каждый ли момент времени данные, к которым обращается приложение, актуально?

Что будет, например, если в процессе обновления нескольких миллионов записей СУБД успела выполнить лишь часть этой работы, и ровно в этот момент приложение выполняет запрос на чтение данных из этой таблицы? Какие данные оно получит? Только обновлённые? Только не обновлённые? Исходный набор, каким он был до начала обновления? Что-то иное? Или же приложение будет вынуждено ждать, пока СУБД закончит обновление всех данных?

Здесь мы приходим к такому явлению как транзакции, которым посвящён отдельный раздел данной книги{374}. Пока же отметим, что от схемы данных (и сопутствующих структур, таких как представления{340}, проверки{347}, триггеры{350}, хранимые процедуры{374}) зависят в том числе и доступные способы взаимодействия приложений с базой данных.

Общий вывод по данной главе: нормализация — это не только способ сделать схему данных неподверженной аномалиям, это ещё и очень ответственная часть процесса проектирования базы данных, требующая повышенного внимания и следования перечню только что рассмотренных правил.

Задание 3.2.d: какие из рассмотренных в данной главе требования нормализации нарушены в модели базы данных «Банк»{408}? Доработайте модель так, чтобы устранить соответствующие недостатки.

Задание 3.2.e: с использованием раздаточного материала{5} доработайте модель базы данных «Банк»{408}, добавив в неё все необходимые комментарии.

Задание 3.2.f: какие проблемы с производительностью{215} базы данных «Банк»{408} можно спрогнозировать на основе её модели? Доработайте модель так, чтобы устранить соответствующие недостатки.

---

160 Если всё же чуть-чуть затронуть техническую часть: существует огромное (фактически, бесконечное) количество алгоритмов инвалидации кэша и способов технической реализации сопутствующих операций. Чем выше требования к производительности системы, тем более нетривиальные решения могут приниматься — вплоть до разработки собственных аппаратно-программных систем.

3.2.3. ПРОЦЕСС НОРМАЛИЗАЦИИ С ПРАКТИЧЕСКОЙ ТОЧКИ ЗРЕНИЯ

Нормализацию можно рассматривать как процесс анализа имеющейся схемы базы данных на предмет выявления зависимостей{174} и ограничений, а также существующих аномалий операций с данными{161} и избыточности.

Фактически, нормализация представляет собой «прочёсывание» схемы базы данных с целью выявления и устранения её недостатков.

Схемы отношений, обладающих выявленными недостатками, будут перерабатываться (как правило — декомпозироваться{180}) в новые схемы, избавленные от этих недостатков.

Таким образом, нормализация предоставляет:

- логическую основу для анализа схем отношений;
- набор нормальных форм как признаков глубины нормализации и инструментов процесса нормализации;

Рассматривая любую схему отношения, можно подвергнуть её серии проверок на предмет того, находится ли она в той или иной нормальной форме. Если необходимая глубина нормализации не достигнута, схему можно декомпозировать{180}.

Важно понимать, что само по себе соответствие некоторой схемы отношения некоей нормальной форме не является признаком хорошо выполненного проектирования — это лишь одно из условий. Во внимание следует принимать как универсальные требования к любой базе данных{12}, так и множество других факторов, зависящих от предметной области, особенностей выбранной технологической платформы и т.д. и т.п.

Также стоит отметить, что на практике нормализацию часто прекращают на уровне 3НФ{252} (иногда — на уровне НФБК{258}), т.к. нормальные формы более высоких порядков зачастую не приносят дополнительных преимуществ схеме базы данных, а лишь усложняют её. Иногда нормализация может быть прекращена и на более низких уровнях (например, на уровне 2НФ{246}) — как правило, с целью обеспечения необходимой производительности операций.

В рассмотренном далее примере мы пройдём весь процесс нормализации от 0НФ до 6НФ, рассматривая формальные доказательства нахождения (или не нахождения) схемы каждого отношения в той или иной нормальной форме.

Критически важно! Как уже не раз было сказано{75}, {174}, все рассуждения о свойствах и особенностях схем отношений, переменных отношений, самих отношений можно проводить **только** в контексте предметной области. Потому наличие или отсутствие той или иной зависимости мы также будем аргументировать требованиями предметной области, которые будем явно указывать.

Да, сами нормальные формы будут рассмотрены далее{240}, однако практика показала, что для понимания и запоминания лучше сначала рассмотреть процесс нормализации, а потом — сами нормальные формы.

Итак…

0НФ, «ужасная схема»

Представим, что в силу каких-то невероятных стечений обстоятельств, нам досталась такая схема отношения (см. рисунок 3.2.3.a) — да, в здравом уме такую схему никто не сделает, но всё же…

Даже сложно себя заставить сказать, что эта схема отношения находится хоть в какой бы то ни было нормальной форме. Но формально — это т.н. «нулевая нормальная форма{280}», которая не предъявляет к схеме отношения вообще никаких требований.

По условиям предметной области нам нужно будет сохранять такие данные о сотруднике как:

- ФИО;
- должность;
- отдел;
- телефоны отдела;
- наличие служебного автомобиля;
- имена и даты рождения всех детей сотрудника.

В настоящий момент, видимо, предполагается это всё хранить в произвольном виде в колонке `data`.

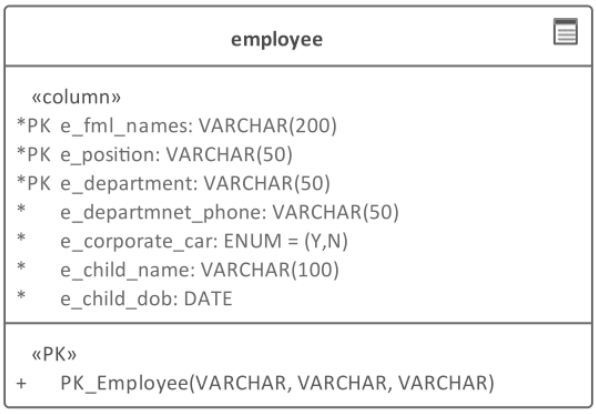
*Рисунок 3.2.3.a — **Схема отношения в 0НФ***

Очевидно, что нас такая ситуация не устраивает, и потому мы начнём нормализацию этой схемы приведением её к 1НФ{241}.

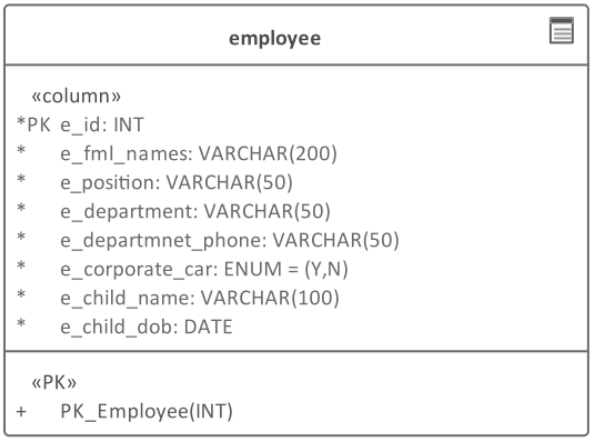
1НФ, «атомизированные атрибуты»

С этого момента и далее мы будем рассматривать два варианта развития событий: с естественными{42} и искусственными{42} первичными ключами.

Первый результат приведения данной схемы отношения к 1НФ{241} показан на рисунке 3.2.3.b.0



Вариант с естественным первичным ключом



Вариант с искусственным первичным ключом

*Рисунок 3.2.3.b — **Схема отношения в 1НФ, шаг 1***

Для варианта с естественным первичным ключом мы вынуждены ввести дополнительное ограничение предметной области: в одном и том же отделе на одной и той же должности не может быть двух людей с одинаковыми ФИО.

Для варианта с искусственным первичным ключом это ограничение можно не вводить. Но! Если это ограничение существует объективно (например, в фирме есть правило, запрещающее в одном и том же отделе на одной и той же должности работать людям с одинаковыми ФИО), нам придётся создать уникальный индекс на полях **`e_fml_names`**, **`e_position`**, **`e_department`**.

По большинству полей их атомарность не вызывает вопросов, но сомнения могут быть относительно поля **`e_fml_names`**, хранящего ФИО.

Как показано в описании 1НФ{241}, не существует никакой «магической формулы», позволяющей определить, атомарно поле или же нет. Потому здесь мы снова прибегнем к требованиям предметной области и постановим, что в процессе использования этой базы данных никогда и ни при каких обстоятельствах не понадобится использовать отдельно фамилию, имя и отчество сотрудника[161], т.е. поле **`e_fml_names`** будем считать атомарным.

А вот поле **`e_department_phone`** — не атомарно. По требованию предметной области у отдела может быть несколько телефонов, т.е. здесь получается классическое нарушение 1НФ{241}, выражающееся в том, что в схеме отношения есть многозначный атрибут (поле **`e_department_phone`** вынуждено хранить много номеров телефонов).

Чтобы исправить эту проблему, мы декомпозируем схему **`employee`** на две (см. рисунок 3.2.3.c).

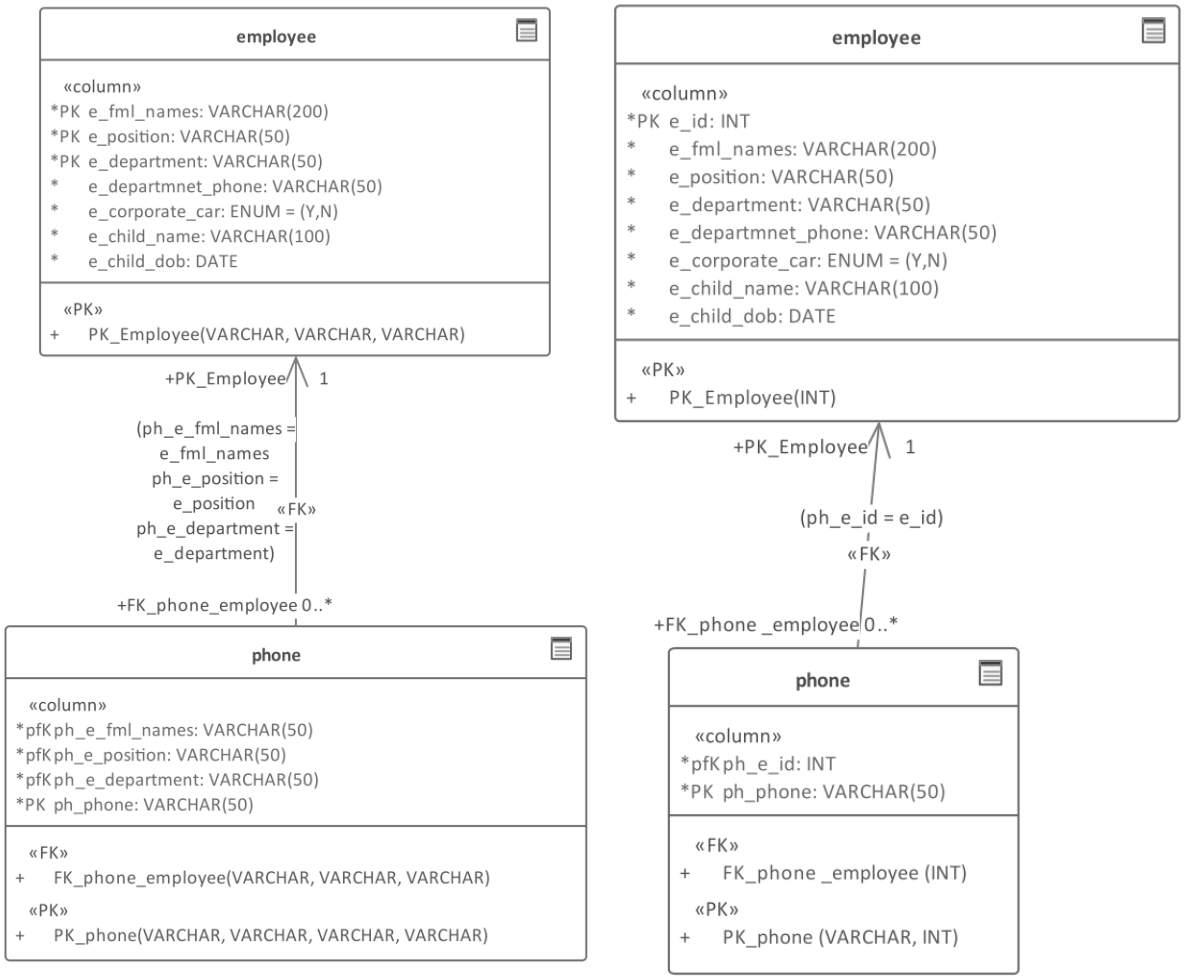
Получившийся результат выглядит крайне сомнительно с точки зрения здравого смысла, но на текущем этапе у нас нет выбора — мы обязаны привязать номер телефона к первичному ключу родительской таблицы (просто к отделу — нельзя, его название не является первичным ключом).

Здесь же сразу видна проблема: мы не можем исключить возможность привязки одного и того же номера телефона к разным отделам (использование триггеров{350} может решить эту проблему, но даже там придётся изобретать не до конца очевидный алгоритм принятия решения о разрешении или запрете операций вставки и модификации телефонов).

Да, мы могли бы временно «забыть» про эту проблему с неатомарностью поля **`e_department_phone`** и вернуться к ней после устранения частичных зависимостей{184}, но в учебном контексте это было бы нарушением последовательности изложения, потому будем пока нервничать из-за этого странного решения.

---

161 Это очень грубое допущение, почти никогда не выполняющееся в реальной жизни. Но здесь оно введено для упрощения дальнейших рассуждений.



Вариант с естественным первичным ключом Вариант с искусственным первичным ключом

*Рисунок 3.2.3.c — **Схемы отношений в 1НФ, шаг 2 (финальный)***

Итак, все требования 1НФ выполнены: все поля атомарны, есть первичные ключи (второе требование не получило широкого распространения при формулировке 1НФ, но раз мы сразу выполнили и его — хуже точно не стало).

2НФ, «устранение частичных зависимостей»

Здесь нам понадобится сильная формулировка 2НФ{246}, т.к. если пользоваться слабой формулировкой, вариант с искусственным ключом уже находится во 2НФ.

Сильная формулировка 2НФ говорит, что переменная отношения должна находиться в 1НФ (это мы уже обеспечили), а также каждый неключевой атрибут{23} переменной отношения должен функционально полно{183} зависеть от любого потенциального ключа{37}.

Какие у нас есть потенциальные ключи? Это уже упомянутая тройка атрибутов `e_fml_names`, `e_position`, `e_department`, которая в варианте с естественным первичным ключом как раз и стала самим первичным ключом, а в варианте с искусственным первичным ключом осталась альтернативным ключом{38}.

Сразу бросается в глаза, что телефоны отдела зависят исключительно от отдела, но никак не зависят от ФИО и должности сотрудника. Да, мы убрали телефоны в отдельное отношение, но проблему это не решило — сейчас у нас просто ошибка в схеме: телефоны «привязаны» к сотруднику, а не к отделу.

И для завершения картины добавим ещё одно требование предметной области: служебные автомобили положены только сотрудникам, занимающим определённые должности.

Итак, у нас получаются две частичные зависимости:
поле **ph_phone** зависит только от поля **e_department**;
поле **e_corporate_car** зависит только от поля **e_position**.

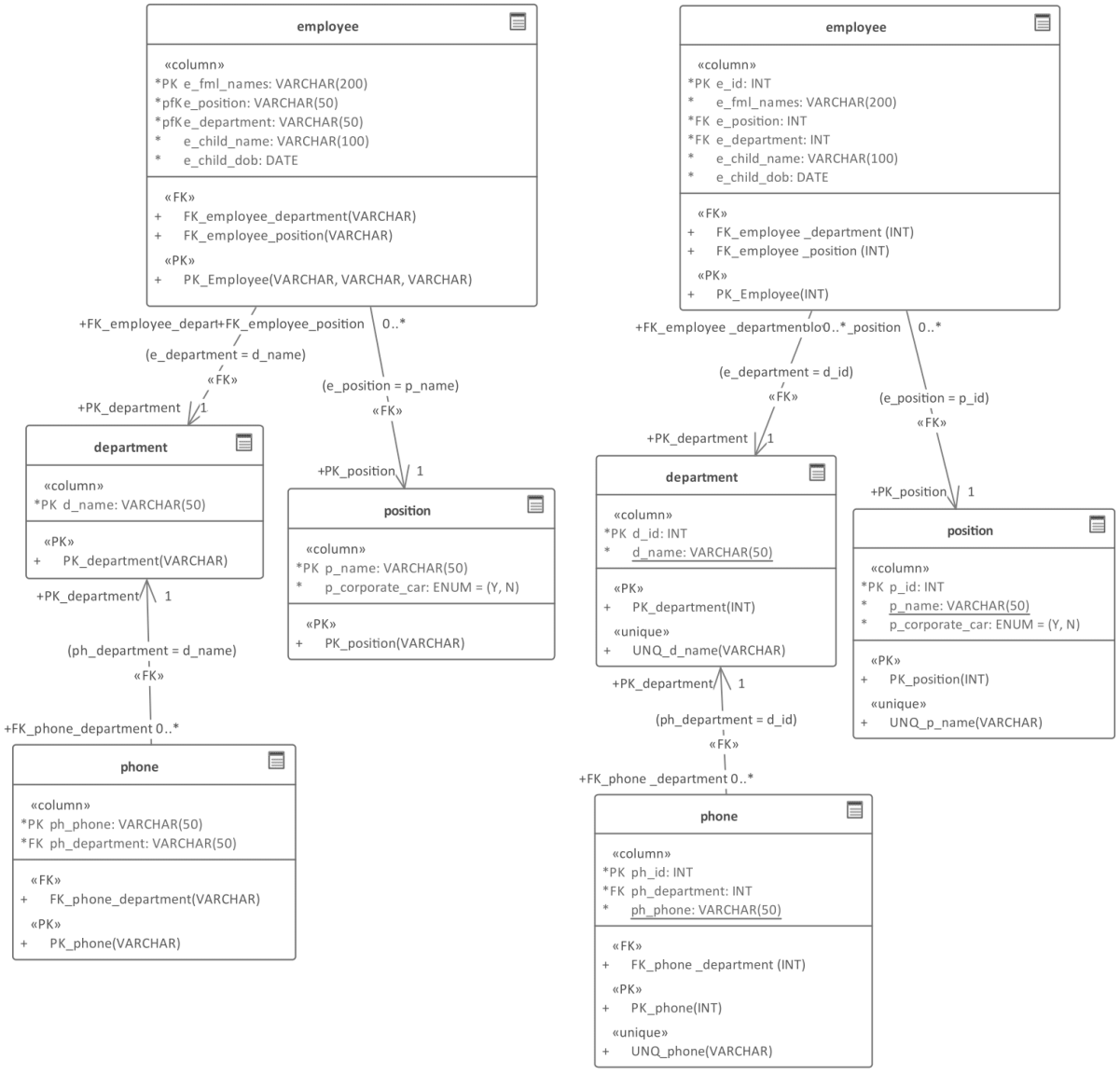
Устраним эти зависимости, декомпозировав схему отношения **employee** на несколько проекций (см. рисунок 3.2.3.d).

В обоих вариантах (с естественным и искусственным первичным ключом) поля, зависящие от части потенциального ключа, были вынесены в отдельные схемы отношений, и таким образом схема отношения **employee** была приведена ко 2НФ.

Полученные схемы отношений **position**, **department**, **phone** также находятся во 2НФ: их атрибуты атомарны (мы не добавляли новых полей, кроме искусственных первичных ключей), а т.к. в них нет составных потенциальных ключей, частичная зависимость неключевого атрибута от потенциального ключа не может существовать здесь по определению (чтобы она могла существовать, у потенциального ключа должны быть части, т.е. он должен быть составным).

Несмотря на то, что вариант с искусственными первичным ключом кажется более громоздким, при большом количестве сотрудников, и небольшом количестве отделов и должностей он позволяет выиграть в производительности и объёме хранимых данных за счёт более компактных внешних ключей в отношении **employee**.

Да, для обеспечения уникальности названий отделов и должностей, а также для обеспечения уникальности телефонных номеров нам пришлось сделать здесь уникальные индексы{109}, но объективно эти отношения будут изменяться крайне редко, а потому потери в производительности не будет.



Вариант с естественным первичным ключом Вариант с искусственным первичным ключом

*Рисунок 3.2.3.d — **Схемы отношений во 2НФ***

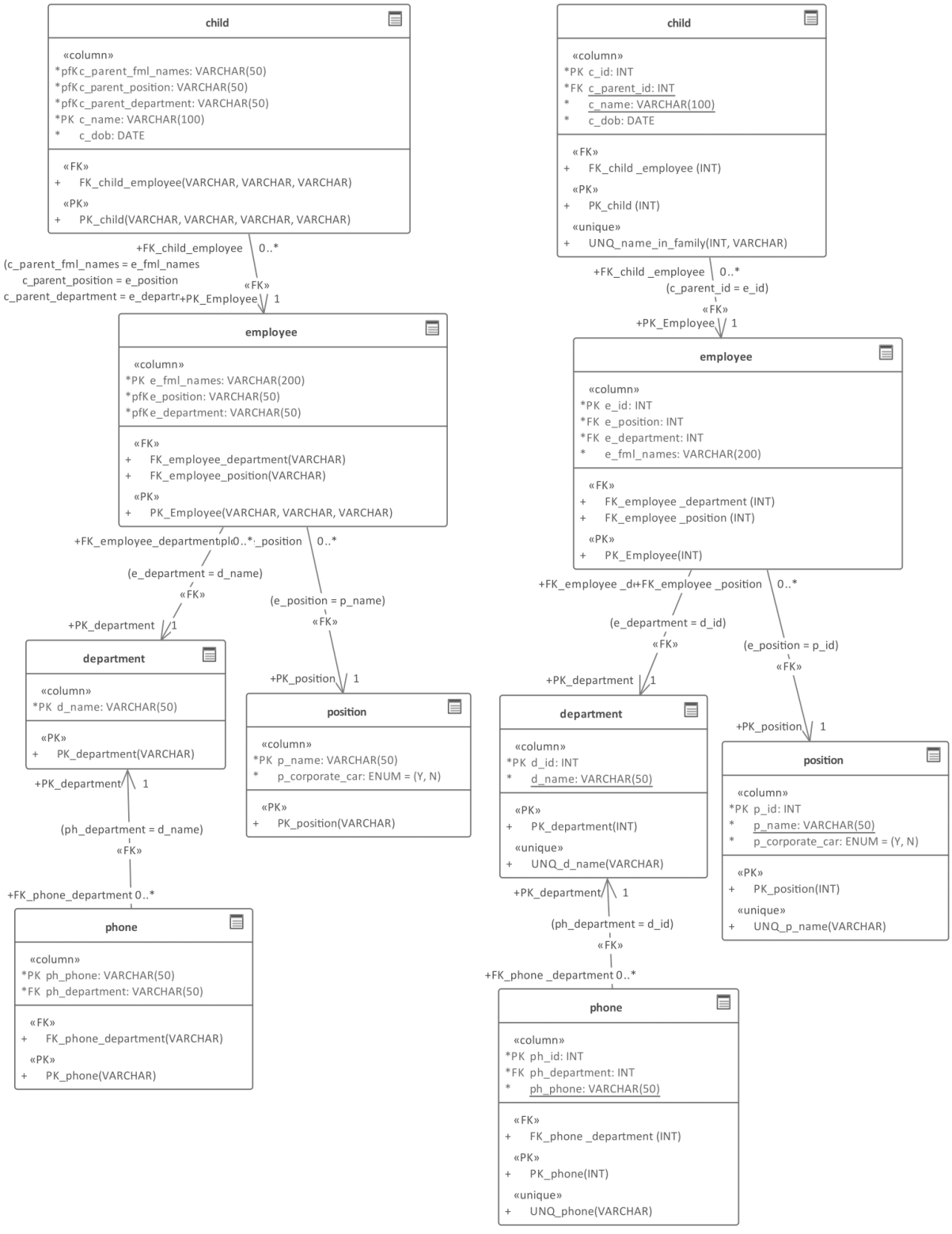
Почему мы не тронули поля `e_child_name` и `e_child_dob`? Потому, что они зависят лишь от всего потенциального ключа целиком (т.е. здесь нет частичной зависимости). Ведь для ответа на вопрос «чей это ребёнок?» нужно гарантированно идентифицировать его родителя, и одной лишь должности, или отдела, или ФИО недостаточно.

Итак, для всех имеющихся схем отношения все требования 2НФ выполнены: схемы отношения находятся в 1НФ, и ни один неключевой атрибут не зависит частично от какого бы то ни было потенциального ключа.

3НФ, «устранение транзитивных зависимостей»

Осталось разобраться со следующей проблемой: день рождения ребёнка зависит от ребёнка, а не от его родителя, т.е. имеется транзитивная зависимость `{первичный ключ}` → `e_child_name` → `e_child_dob`.

Снова декомпозируем схему отношения `employee` (см. рисунок 3.2.3.e).



Вариант с естественным первичным ключом Вариант с искусственным первичным ключом

*Рисунок 3.2.3.e — **Схемы отношений в 3НФ***

Полученная схема отношения **`child`** находится в 3НФ{252}, т.к.:

- 1НФ: все её атрибуты атомарны;
- 2НФ: нет частичных зависимостей какого бы то ни было неключевого атрибута от первичного ключа:
  - в варианте с естественным первичным ключом есть только один неключевой атрибут **`c_dob`**, который определяется целиком всем первичным ключом (чтобы узнать дату рождения ребёнка, нужно знать **и** чей это ребёнок, **и** что это за ребёнок);
  - в варианте с искусственным первичным ключом этот ключ состоит из одного поля, а альтернативный ключ из полей **`c_parent`** и **`c_name`** также целиком необходим для определения даты рождения ребёнка.
- 3НФ: поскольку кроме первичного (и альтернативного в варианте с искусственным первичным) ключа и единственного неключевого атрибута в отношении нет других атрибутов, по определению транзитивная зависимость не может существовать (т.к. для её существования необходим третий, «промежуточный» атрибут).

Аналогичные рассуждения в полной мере применимы к остальным схемам отношений в полученной модели базы данных. Таким образом, вся модель находится как минимум в 3НФ.

НФБК, 4НФ и т.д.

Полученная схема базы данных (сравните результат на рисунке 3.2.3.e с начальной ситуацией на рисунке 3.2.3.a) уже вполне пригодна к использованию. Но ничто не мешает нам проверить «максимальную глубину» её нормализации.

НФБК{258} требует, чтобы в каждой нетривиальной{191} функциональной зависимости {X} → A множество {X} являлось суперключом{36} . Ранее при приведении к 3НФ мы уже разобрались со всеми неключевыми атрибутами, потому сейчас посмотрим на ключевые.

В варианте с естественным первичным ключом теоретически это правило могло бы быть нарушено в отношениях **`child`** и **`employee`** (в них есть множества, состоящие из нескольких элементов и являющиеся детерминантами функциональных зависимостей), но на самом деле нарушения нет, т.к. зависимые атрибуты тоже являются частями определяющего их множества, т.е. зависимость является тривиальной.

В варианте с искусственным первичным ключом подобный вопрос актуален только для отношения **`child`**, но и там либо зависимость является тривиальной (для множества **`c_parent_id`** и **`c_name`**, определяющего свои компоненты), либо выполняется требование НФБК о том, что детерминант зависимости должен быть суперключом (для зависимости {**`c_parent_id`**, **`c_name`**} → **`c_id`**).

Итак, все отношения полученной модели базы данных находятся в НФБК.

4НФ{263} требует, чтобы для любой существующей в переменной отношения нетривиальной многозначной{193} зависимости X ↠ Y множество X было суперключом.

Но ни в одной переменной отношения в нашей базе данных нет многозначных зависимостей, т.е. условие 4НФ по определению не может быть нарушено.

Итак, все отношения полученной модели базы данных находятся в 4НФ.

5НФ{268} требует, чтобы для каждой существующей в переменной отношения нетривиальной зависимости соединения{196} JD(R_1, R_2, …, R_n) каждый набор атрибутов R_i являлся суперключом{36} исходной переменной отношения.

Мы не можем выполнить декомпозицию без потерь{180} ни одной из имеющихся схем отношения, не включив в каждую из проекций первичный и/или потенциальный ключи. А множество атрибутов, содержащее в своём составе первичный и/или потенциальный ключ, по определению является суперключом.

Итак, все отношения полученной модели базы данных находятся в 5НФ.

ДКНФ{273} требует, чтобы все ограничения и зависимости, существующие в переменной отношения, являлись следствиями ограничений доменов{22} и ключей{35} и не существовало никаких «скрытых» правил, вытекающих из чего бы то ни было другого.

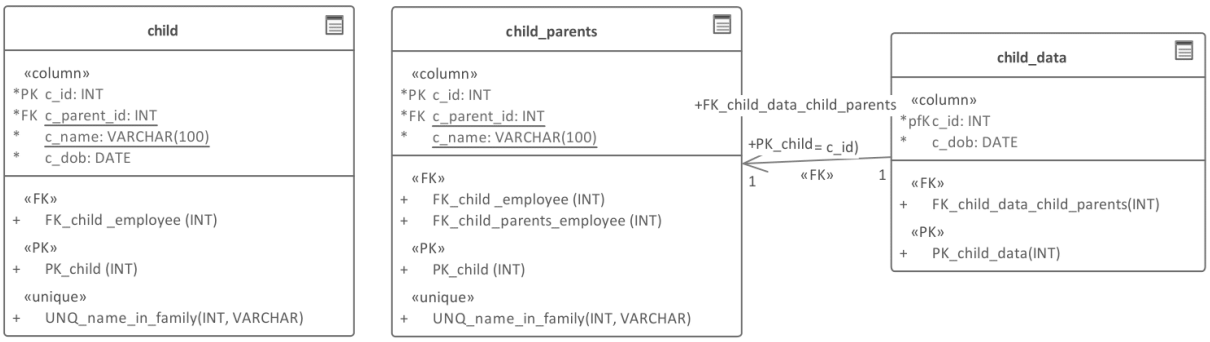
Здесь доказательство будет немного странным, но всё же: постановим, что все требования предметной области мы выполнили и отразили их в структуре модели базы данных (к слову, созданные уникальные индексы в варианте с искусственными первичными ключами следуют как раз этой цели: мы гарантируем уникальность значений альтернативных ключей). Тогда требования ДКНФ можно считать выполненными.

Итак, все отношения полученной модели базы данных находятся в ДКНФ.

6НФ{277} требует, чтобы переменная отношения не допускала в принципе никакой декомпозиции без потерь{180}, т.е. любые имеющиеся в ней зависимости соединения{196}, {204} являлись тривиальными{196}, {204}.

Для варианта с естественными первичными ключами это требование выполняется, т.к. в каждой имеющейся схеме отношения есть лишь одно неключевое поле (в итоге хотя бы одна проекция обязана иметь в своём составе первичный ключ и это неключевое поле, т.е. проекция эквивалентна исходной схеме, т.е. зависимость соединения является тривиальной).

Для варианта с искусственными первичными ключами дальнейшая декомпозиция некоторых отношений возможна, но… бессмысленна. Продемонстрируем это на примере схемы отношения **`child`** (рисунок 3.2.3.f).



До декомпозиции После декомпозиции

*Рисунок 3.2.3.f — **Бессмысленная декомпозиция схемы отношения** `child`*

Такая декомпозиция возможна технически, но не принесёт ничего, кроме необходимости выполнения дополнительных действий при любых операциях с данными.

Итак, все отношения полученной модели базы данных для варианта с естественными первичными ключами находятся в 6НФ, а отношения **`child`**, **`position`** и **`phone`** для варианта с искусственными первичными ключами не находятся в 6НФ (но как было только что показано — это и не нужно).

В завершении этой главы отметим, что с приобретением некоторого опыта вы начнёте формировать модели баз данных, сразу находящиеся как минимум в 3НФ — с какого-то момента это происходит просто инстинктивно.

Что до формального анализа некоей модели с целью выяснения того, достаточно ли она нормализована, то намного проще и логичнее делать это через продумывание запросов: если вы обнаруживаете ситуацию, в которой «что-то идёт не так» (скорее всего — проявляется аномалия операций с данными{161} или просто получаются неудобные, нелогичные, сложные запросы), значит, стоит рассмотреть варианты переработки имеющихся схем отношений.

Иногда эта переработка будет связана с нормализацией (или денормализацией), но не реже вы просто будете обнаруживать ошибки в модели базы данных, неадекватность предметной области и прочие недостатки, которые можно исправить, даже не затрагивая процесс нормализации.

С нормализацией на этом делаем паузу и рассмотрим обратный процесс — денормализацию, чему и будет посвящена следующая глава.

Задание 3.2.g: в какой нормальной форме находится каждая схема отношения базы данных «Банк»{408}? Стоит ли изменить эту нормальную форму? Если вы считаете, что «да», внесите соответствующие правки в модель.

Задание 3.2.h: существуют ли в базе данных «Банк»{408} схемы отношений, дальнейшая нормализация которых невозможна? Аргументируйте своё мнение.

Задание 3.2.i: существуют ли в базе данных «Банк»{408} схемы отношений, дальнейшая нормализация которых возможна, но по каким-то причинам нежелательна? Аргументируйте своё мнение.

3.2.4. ДЕНОРМАЛИЗАЦИЯ

Как было отмечено ранее{223}, процесс нормализации часто прекращают на уровне 3НФ{252} или НФБК{258}, т.к. более глубокая нормализация либо не требуется, либо даже вредит (в первую очередь приводя к усложнению запросов и снижению производительности).

Такое «досрочное прекращение» нормализации тоже можно какой-то мере считать денормализацией, но, если говорить более строго, определение денормализации выглядит следующим образом.

Денормализация (denormalization[162]) — процесс приведения схем отношений к более низкой нормальной форме путём их объединения.

Упрощённо: результат JOIN'а двух и более таблиц сохраняется как одна новая таблица.

В общем случае можно выделить четыре вида денормализации:
- объединение схем отношений;
- создание кэширующих схем отношений или отдельных атрибутов;
- создание агрегирующих схем отношений или отдельных атрибутов;
- создание постоянных (материализованных) представлений{340}.

И пусть к «классической денормализации» относится только первый вариант, сейчас мы последовательно рассмотрим их все.

Объединение схем отношений

Представим, что в некоторой вымышленной организации существует огромное количество отделов, у каждого из которых есть огромное количество телефонов. Мы сначала решили использовать схему базы данных, представленную на рисунке 3.2.3.e{229}, но очень быстро столкнулись с тем, что выполнение `JOIN`'а таблиц `phone` и `department` занимает недопустимо большое с точки зрения заказчика время.

Одним из вариантов решения данной проблемы будет объединение этих таблиц в одну, как показано на рисунке 3.2.4.a.

Теперь все телефоны отдела хранятся в той же самой таблице (даже в той же самой записи), где и информация о самом отделе, а потому доступны в рамках запроса к одной таблице, что решило проблему «недопустимо долго выполняющегося `JOIN`'а».

Но такое решение породило новую проблему: теперь схема отношения department не находится даже в 1НФ{241} (т.к. у неё есть многозначный атрибут), и мы будем вынуждены реализовывать (на уровне СУБД или на уровне работающих с ней приложений) специальный алгоритм как минимум по обновлению поля `d_phones`.

Отметим также, что поиск отдела, которому принадлежит некий указанный телефон, теперь также становится более сложной и дольше выполняющейся задачей.

Уже на этом примере видно, что универсального идеального решения не существует: выигрывая в чём-то одном, мы жертвуем чем-то другим.

---

162 **Denormalization** is the process of storing the join of higher normal form relations as a base relation, which is in a lower normal form. («Fundamentals of Database Systems», Ramez Elmasri, Shamkant Navathe)

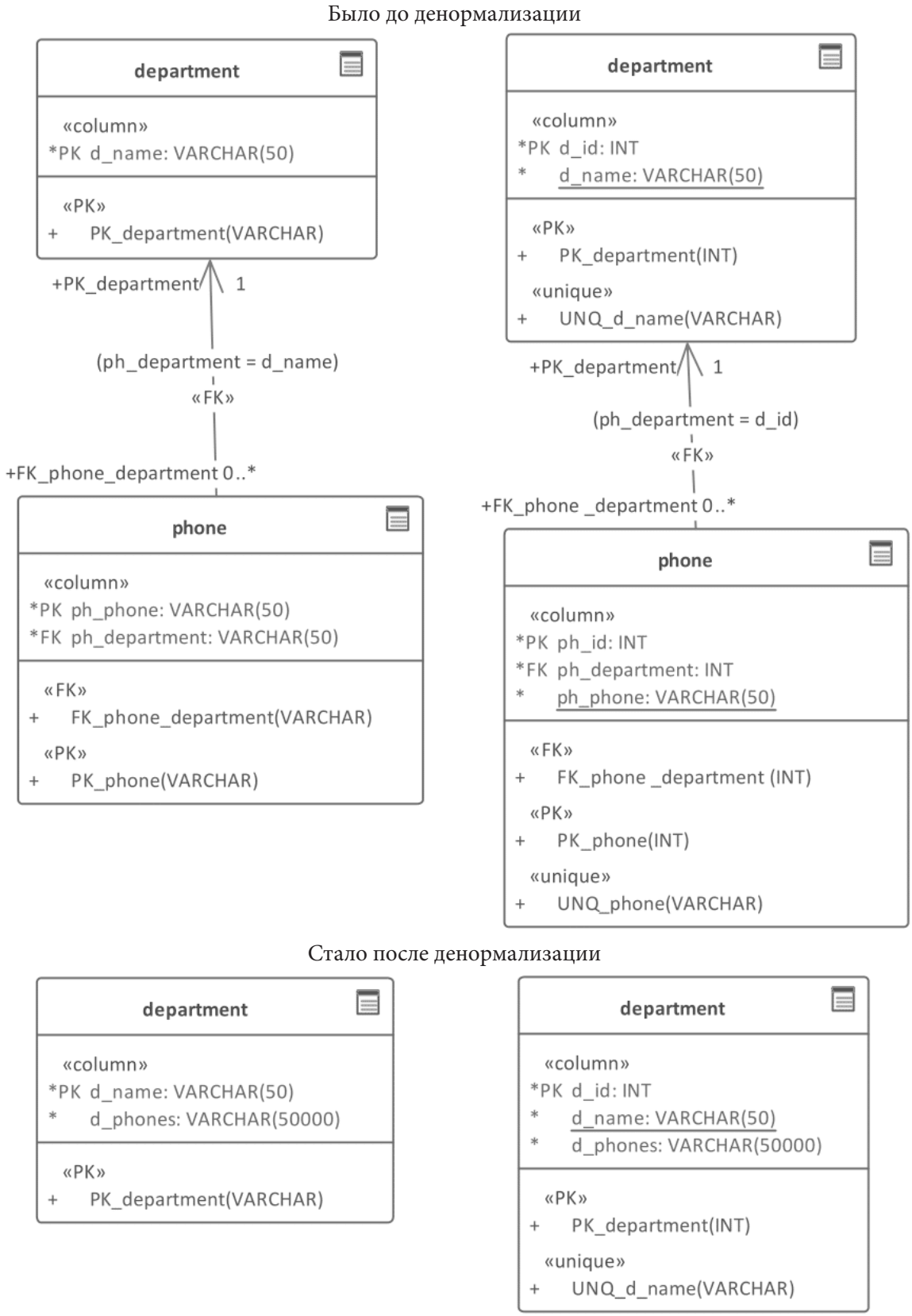


*Рисунок 3.2.4.а — **Денормализация через объединение схем отношений***

Создание кэширующих схем отношений или отдельных атрибутов

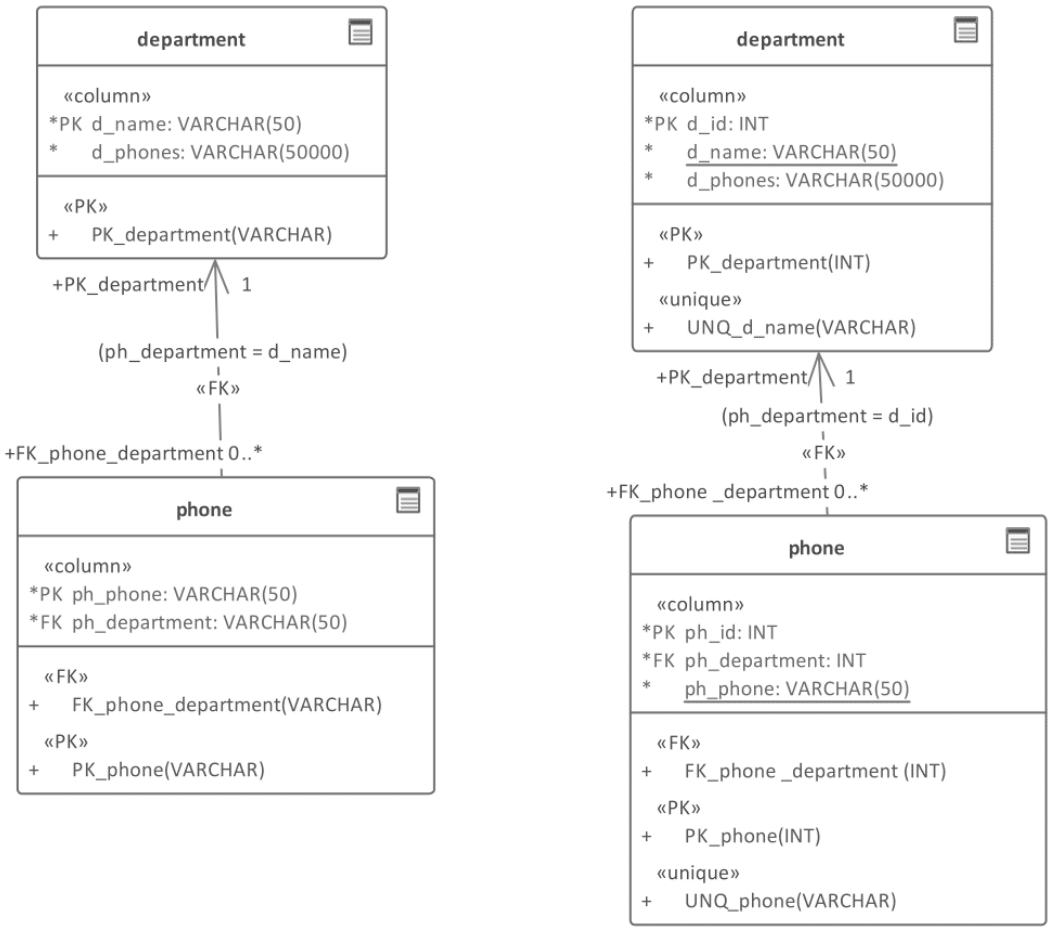
Только что отмеченную проблему с нетривиальным обновлением поля **d_phones** можно решить достаточно элегантным способом: данные в нём можно рассматривать лишь как временное быстрое хранилище (кэш) данных, а долговременное хранение осуществлять (как и прежде) в отдельной таблице.

Такая схема базы данных показана на рисунке 3.2.4.b.

При таком подходе все операции по добавлению, редактированию, удалению телефонных номеров выполняются по-прежнему с использованием отношения phone. Равно как и быстрый поиск отдела, которому принадлежит указанный телефон, тоже можно выполнять с использованием отношения phone.

А поле **d_phones** отношения **department** должно автоматически обновляться, «собирая» в себя все телефоны отдела, при изменениях отношения **department**. Такое автоматическое обновление можно реализовать с использованием триггеров{350} (также см. практические примеры в соответствующей книге[163]).

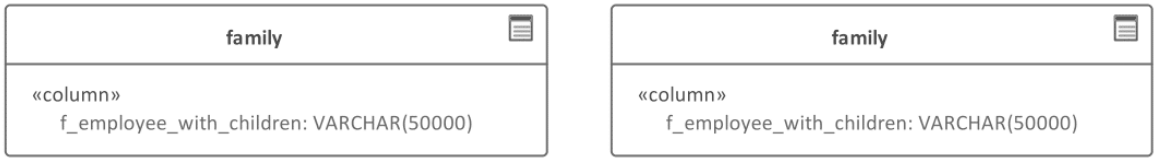
Если пойти на шаг дальше, от кэширующих атрибутов мы перейдём к кэширующим отношениям. Допустим, нам зачем-то нужно мгновенно получать список ФИО сотрудников со списком детей каждого сотрудника. Мы можем создать отдельное отношение **family**, как показано на рисунке 3.2.4.c.



Вариант с естественным первичным ключом Вариант с искусственным первичным ключом

*Рисунок 3.2.4.b — **Пример денормализации через создание кэширующего атрибута***

163 «Работа с MySQL, MS SQL Server и Oracle в примерах» (С.С. Куликов), раздел 4.1 [http://svyatoslav.biz/database_book/]



Вариант с естественным первичным ключом Вариант с искусственным первичным ключом

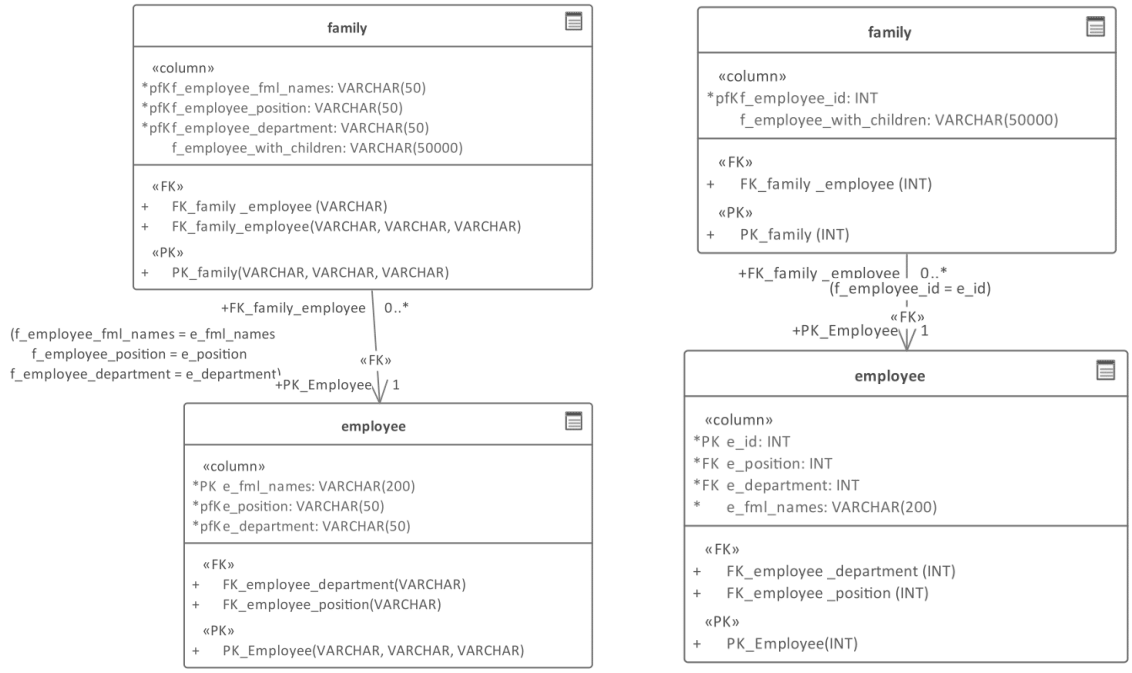
*Рисунок 3.2.4.c — **Первый пример денормализации через создание кэширующего отношения***

Легко заметить, что в данном случае в вариантах с естественным и искусственным первичными ключами нет никакой разницы — мы просто добавили таблицу, в каждой строке которой будет храниться информация об одном сотруднике и всех его детях. Здесь даже нет первичного ключа.

Такое отношение находится в 0НФ{280}, т.е. нарушает все мыслимые и немыслимые правила проектирования баз данных, но оно вполне служит выполнению возложенной на него задачи.

В решении, представленном на рисунке 3.2.4.c мы будем вынуждены обновлять с помощью триггеров целиком все данные в таблице **`family`** каждый раз, когда изменения вносятся в таблицы **`employee`** или **`child`**, т.к. информация о том, «где какой сотрудник» и «где чей ребёнок» у нас нигде не хранится.

Но мы можем немного сократить потенциальные трудозатраты СУБД. Добавим в таблицу **`family`** внешний ключ на таблицу **`employee`** (см. рисунок 3.2.4.d).



Вариант с естественным первичным ключом Вариант с искусственным первичным ключом

*Рисунок 3.2.4.d — **Второй пример денормализации через создание кэширующего отношения***

Теперь при изменении таблиц `employee` и/или `child`, мы можем выяснить для какого конкретно сотрудника нужно обновить данные в таблице `family`, т.е. мы можем обновлять только отдельную запись, а не таблицу целиком.

Такое решение поднимает ещё два важных нюанса:

- обратите внимание, насколько компактнее и логичнее выглядит вариант с искусственным первичным ключом — это ещё один важный аргумент в пользу использования именно таких первичных ключей;
- нужно ли в таблице `family` создавать первичный ключ (или хотя бы просто индекс) на идентификаторе сотрудника — вопрос открытый: это зависит от конкретной СУБД, конкретного метода доступа{33} и многих других параметров, т.е. однозначного ответа не существует (хотя интуитивно хочется создать первичный ключ или уникальный индекс, чтобы хотя бы ускорить операцию «поиска того сотрудника, данные о котором нужно обновить»).

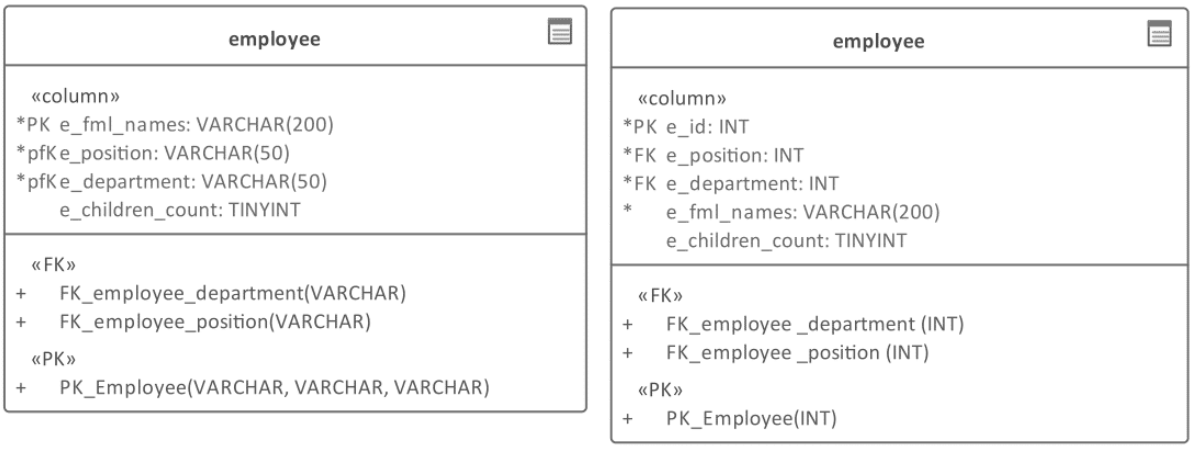
Казалось бы, дальше денормализовать схему базы данных уже некуда, мы уже «упёрлись в дно», достигнув 0НФ{280}. Но… можно пойти и ещё дальше.

Создание агрегирующих схем отношений или отдельных атрибутов

Разница между кэшированием и агрегацией[164] состоит в том, что на основе кэша (пусть и не всегда, пусть теоретически) есть шанс восстановить исходные данные (да, в реальности это не требуется, но возможность есть), а на основе агрегированных данных исходную информацию невозможно восстановить в принципе. И тем не менее агрегированные данные очень полезны.

Продолжим пример с сотрудниками и их детьми. Допустим, нам зачем-то нужно мгновенно узнавать количество детей у любого сотрудника (т.е. выполнение `JOIN` с `COUNT()` является недопустимо долгой операцией).

Мы можем заранее вычислять искомые значения (да, снова триггерами{350}) и сохранять их в таблице `employee` в поле `e_children_count`, как показано на рисунке 3.2.4.



Вариант с естественным первичным ключом — Вариант с искусственным первичным ключом

*Рисунок 3.2.4.e — **Пример денормализации через создание агрегирующего атрибута***

164 Поскольку результат агрегации, фактически, хранится в кэширующем поле или отношении, этой терминологической разницей очень часто пренебрегают. Она на самом деле несущественна. В обоих случаях «некие данные как-то собираются и хранятся в удобном виде в месте, откуда их можно в готовом виде очень быстро извлечь».

Агрегирующие отношения, как правило, применяются для сохранения большого количества «разрозненной» информации. Например, нам для корпоративного портала нужно в некий виджет мгновенно отдавать данные о том, сколько у нас сотрудников, сколько у сотрудников всего детей, скольки сотрудникам положены служебные автомобили.

Объективно эту информацию нельзя поместить ни в одно из имеющихся отношений, т.е. нужно создать новое. И создать его можно двумя способами, которые мы условно назовём «горизонтальным» и «вертикальным».

«Горизонтальный» способ применяется в случаях, когда нужно агрегировать небольшой неизменный набор данных — см. рисунок 3.2.4.f. В такой таблице всегда будет ровно одна строка, каждый столбец которой содержит искомые данные.

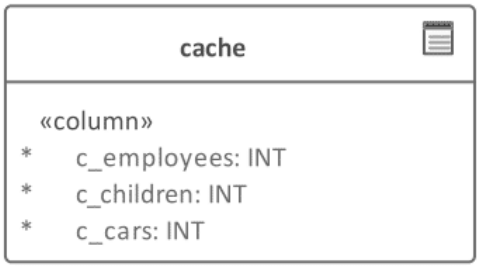


Схема отношения

c_employees	c_children	c_cars
257	342	125

Пример данных

*Рисунок 3.2.4.f — «**Горизонтальное» агрегирующее отношение***

«Вертикальный» способ применяется в случаях, когда нужно агрегировать большой и/или изменяющийся набор данных — см. рисунок 3.2.4.g. В такой таблице всегда будет ровно два столбца (наименование параметра, значение параметра), а каждая её строка будет хранить одну пару вида «ключ-значение».

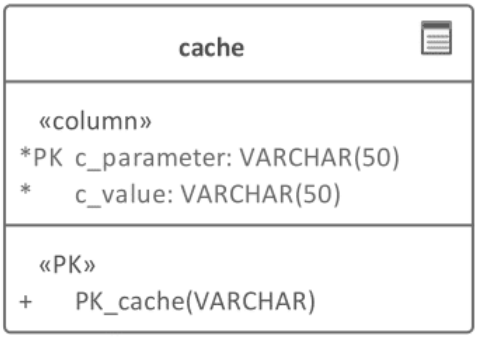


Схема отношения

c_parameter	c_value
employees	257
children	342
cars	125

Пример данных

*Рисунок 3.2.4.g — «**Вертикальное» агрегирующее отношение***

Несмотря на кажущуюся большую универсальность, у «вертикального» варианта есть один серьёзный недостаток: мы не можем для разных параметров указывать разные типы данных. Потому, как правило, строковый тип данных выбирают как наиболее универсальный.

Создание постоянных (материализованных) представлений

Это решение является частным случае создания кэширующих и агрегирующих отношений для СУБД, поддерживающих т.н. постоянные (материализованные) представления{340}.

Подробнее о самом механизме будет сказано в соответствующем разделе{340}, а пока отметим суть: данные, попадающие в представление, автоматически сохраняются СУБД в виде таблицы, и столь же автоматически обновляются по указанному алгоритму.

Такой подход позволяет избежать создания триггеров{350}, однако во многих случаях приводит либо к повышенной нагрузке на СУБД (в случае, если данные в постоянном представлении обновляются тогда, когда обновление не требуется), либо к ситуации устаревания кэша (когда данные в постоянном представлении не успели обновиться, хотя реальные данные уже изменились).

В этом плане триггеры и классические кэширующие и агрегирующие представления дают больше свободы действий, но ничто не мешает совмещать данные подходы, если выбранная вами СУБД это позволяет.

В завершении данной главы отметим два важных факта:

- стоит отличать денормализованную схему базы данных (которая была сначала нормализована и потом денормализована) от просто недостаточно нормализованной схемы: первое — хорошо, а второе — просто пример плохого дизайна;
- денормализация, принося свои плюсы, всё же имеет цену: требуется создавать и поддерживать дополнительные структуры базы данных, жертвовать объёмом хранимых данных и производительностью некоторых операций.

Теперь нам остаётся рассмотреть лишь сами нормальные формы. Приступим…

Задание 3.2.j: существуют ли в базе данных «Банк»{408} схемы отношений, денормализация которых привела бы к повышению производительности и при этом не привела к появлению аномалий{161} в работе с данными? Аргументируйте своё мнение.

Задание 3.2.k: стоит ли в базу данных «Банк»{408} добавить какие бы то ни было агрегирующие схемы отношений? Если вы считаете, что «да», внесите в модель соответствующие правки.

Задание 3.2.l: стоит ли в базу данных «Банк»{408} добавить какие бы то ни было материализованные представления? Если вы считаете, что «да», внесите в модель соответствующие правки.

3.3. НОРМАЛЬНЫЕ ФОРМЫ

Если вы по какой-то причине пропустили раздел, посвящённый теории зависимостей{174}, и сейчас сомневаетесь, что в достаточной мере знакомы с математическими основами нормальных форм, рекомендуется всё же освежить в памяти соответствующий материал{174}.

В различной литературе вы можете встретить формулировки определения нормальных форм, начинающиеся со слов «схема отношения находится…», «переменная отношения находится…», «отношение находится…» (подробнее про разницу этих понятий см. здесь{24}).

Если говорить совершенно корректно с научной точки зрения, стоит использовать в определении нормальных форм термин «переменная отношения» (см. первоисточник[165]).

Однако далее мы в некоторых случаях совершенно осознанно будем использовать термин «отношение» вместо «переменная отношения», чтобы избежать дополнительных нетривиальных пояснений и упростить подачу материала. Тем более, что во всех примерах приведены конкретные *отношения*, пусть и являющиеся значениями *переменной отношения*.

Прежде, чем рассматривать конкретные нормальные формы, дадим общее определение.

Нормальная форма (НФ, NF[166]) — определённое свойство переменной отношения, которое зависит от вида нормальной формы и обеспечивается путём нормализации{211}.

Упрощённо: одна из перечисленных ниже нормальных форм (да, это определение выглядит несколько «рекурсивно», но в подавляющем большинстве первоисточников чистого определения «нормальной формы» не приведено вовсе).

Теперь переходим к конкретике.

---

165 C.J. Date, "An Introduction to Database Systems", 8th edition. Глава 3.3 ("Relations and relvars").

166 **Normal form** of a relvar: see first normal form; second normal form; etc. («The New Relational Database Dictionary», C.J. Date)

3.3.1. ПЕРВАЯ НОРМАЛЬНАЯ ФОРМА

Исторически первая нормальная форма была создана для запрета создания многозначных и составных атрибутов и их комбинаций (т.н. «таблиц внутри таблиц»). Поскольку сегодня ни одна реляционная СУБД не позволит создать «таблицу внутри таблицы», акцент в определении первой нормальной формы смещается в сторону более глубокой трактовки понятия «атомарность».

Переменная отношения находится в **первой нормальной форме** (1НФ, 1NF[167]) тогда и только тогда, когда каждый атрибут отношения содержит строго одно атомарное значение[168].

Упрощённо: каждый атрибут отношения атомарен (т.е. СУБД не должна оперировать никакой отдельной частью атрибута).

Рассмотрим примеры нарушения первой нормальной формы (см. рисунок 3.3.а).

Как только что было отмечено, ни одна современная реляционная СУБД не позволит в качестве типа поля таблицы указать «таблица», т.е. создать «таблицу внутри таблицы». И пусть объектно-реляционные СУБД позволяют в качестве значений атрибута использовать сложные структуры (фактически, объекты), мы вернёмся к классическим реляционным базам данных и рассмотрим подробнее случаи многозначных и составных атрибутов.

На рисунке 3.3.а многозначный атрибут `s_mark` представляет собой список значений оценок. Очевидно, что при выполнении таких типичных операций как поиск минимальной, максимальной и средней оценки СУБД будет вынуждена анализировать содержимое данного атрибута в каждой записи — иными словами, атрибут не атомарен.

---

167 A relvar is in **1NF** if and only if in every legal value of that relvar every tuple contains exactly one value for each attribute. (C.J. Date, "An Introduction to Database Systems", 8th edition)

168 Некоторые авторы явно добавляют к этому определению необходимость наличия у отношения первичного ключа (собственно, потому 1НФ, 2НФ, 3НФ, НФБК и называются «нормальными формами на основе первичных ключей»), но в подавляющем большинстве первоисточников этого требования нет. В любом случае, такие тонкости представляют скорее научный, а не практический интерес.

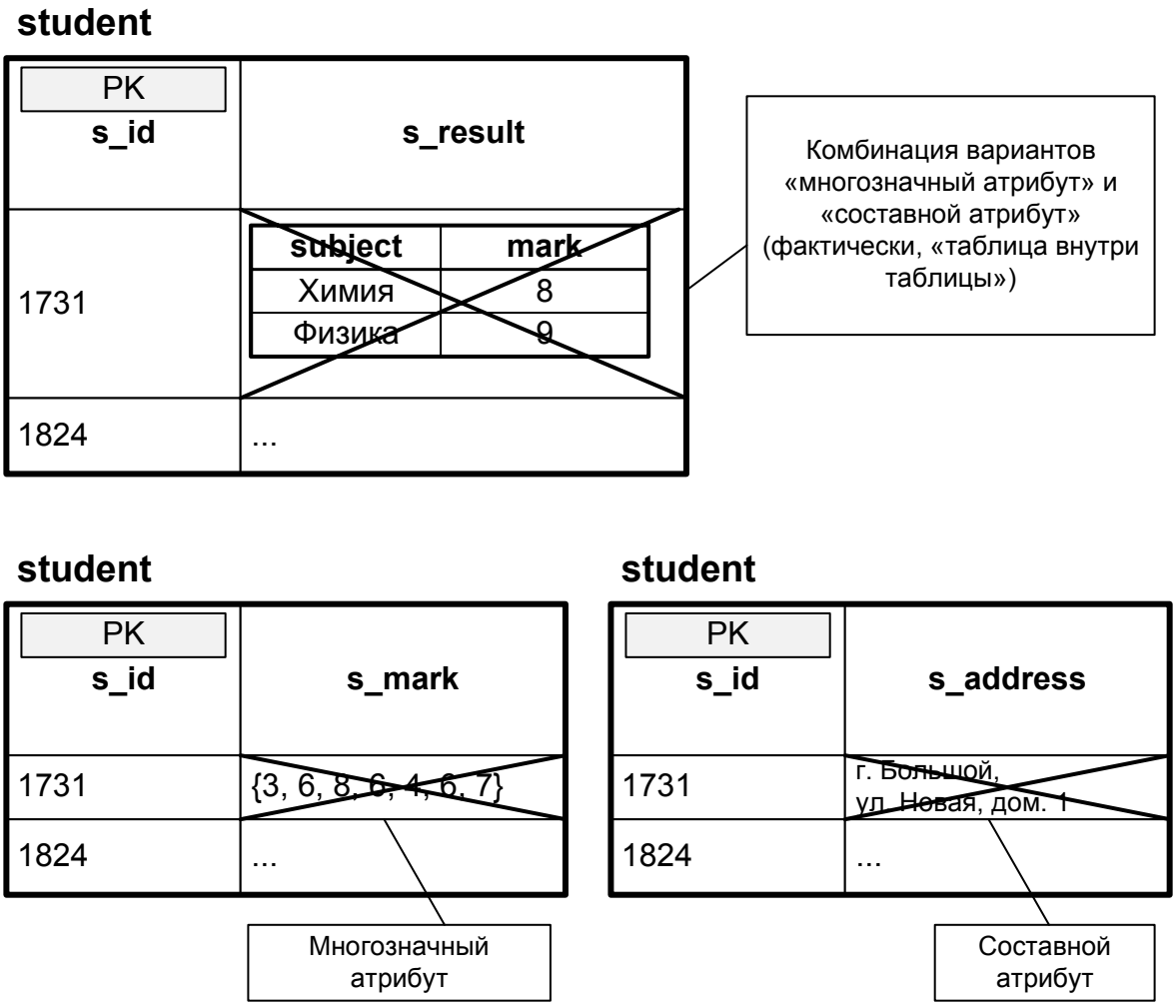


*Рисунок 3.3.a — **Нарушения первой нормальной формы***

Для приведения к 1НФ отношения с многозначными атрибутами необходимо вынести каждый такой атрибут в отдельное отношение и связать его с исходным отношением связью «один ко многим[57]». Результат выполнения такой операции показан на рисунке 3.3.b.

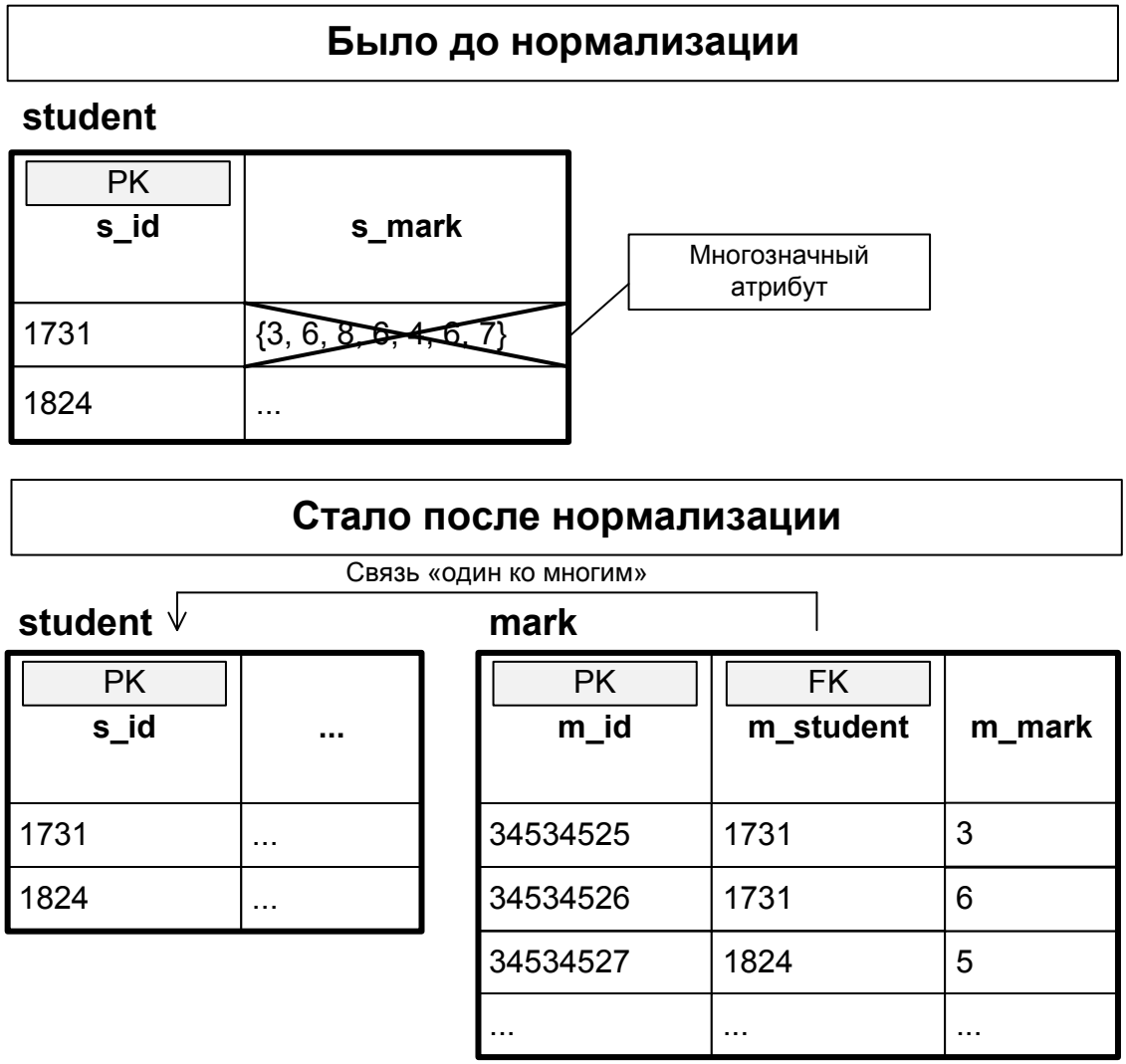


*Рисунок 3.3.b — **Приведение к 1НФ отношения с многозначными атрибутами***

Если с многозначными атрибутами всё просто и очевидно, то с составными вопрос становится почти философским: как мы можем понять, что атрибут уже «достаточно атомарен»? В примере на рисунке 3.3.a всё достаточно тривиально: да, можно город, улицу и номер дома поместить в отдельные поля — результат выполнения такой операции показан на рисунке 3.3.c.

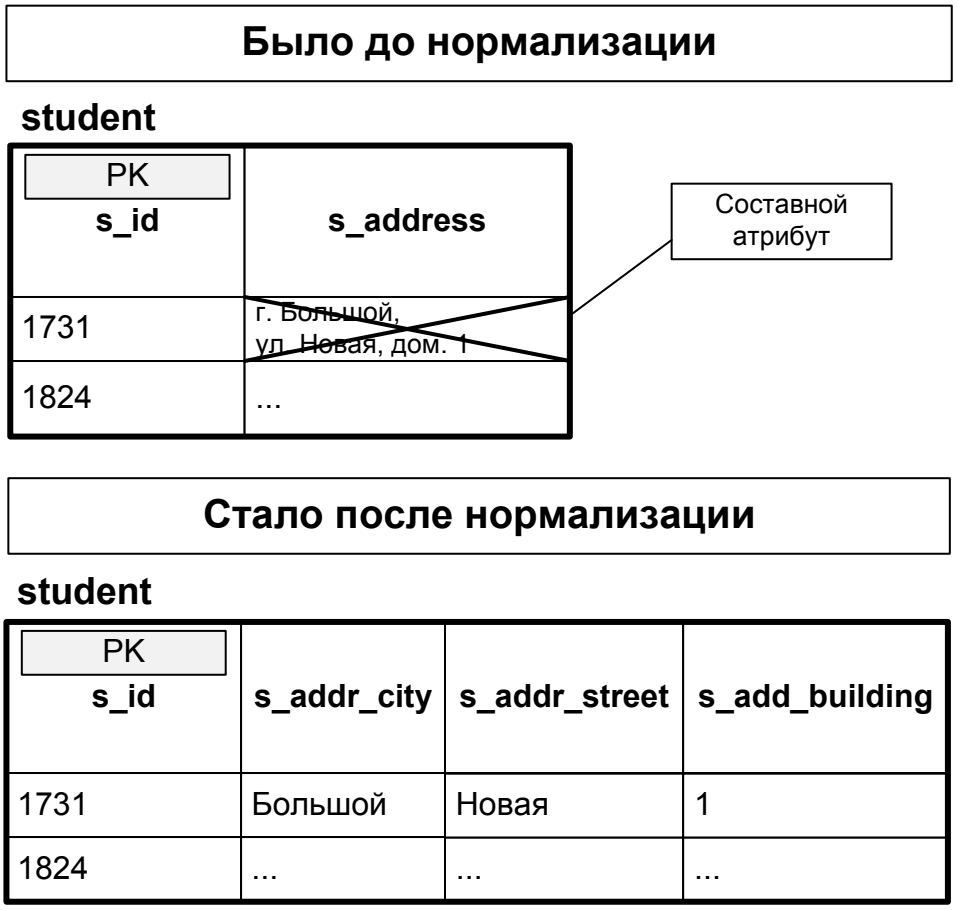


*Рисунок 3.3.c — **Приведение к 1НФ отношения с составными атрибутами***

И теперь неожиданный вопрос: а нужно ли было выполнять эту операцию? Действительно ли атрибут `s_address` не был атомарным? Как правило, здесь большинство считает, что не был, и потому операция его разбиения на три отдельных атрибута оправдана.

А как насчёт следующих случаев:

- Стоит ли хранить в отдельных полях код оператора и номер телефона?
- Стоит ли хранить в отдельных полях имя пользователя и домен для адреса e-mail?
- Стоит ли хранить в отдельных полях год, месяц и день? Если да, то, может быть, стоит сохранить ещё и день недели? И номер недели в году? И признак того, выходной ли это?
- А для времени часы, минуты и секунды (а также их дробную часть) стоит хранить в отдельных полях?
- А серию и номер паспорта стоит хранить в отдельных полях?

Таких вопросов, т.е. примеров из предметных областей, можно придумать очень много. Их объединяет одно: сомнение в целесообразности разбиения исходного значения на составные части. Ведь если нам «не нужны» отдельные части (мы не заставляем СУБД их анализировать) такое разбиение как минимум бессмысленно, а как максимум ещё и вредно (нам придётся постоянно объединять эти фрагменты в единое целое при выполнении операций с данными).

Если подойти к этой ситуации бездумно, мы доведём её до абсурда и начнём каждый бит данных хранить в отдельном поле таблицы — теперь уж точно «атомарнее не бывает».

Но с практической точки зрения всё не так плохо: мы должны оценивать количество и сложность операций, при выполнении которых мы будем анализировать часть значения атрибута. Конечно, в идеале количество таких операций равно нулю, и тогда наше отношение полностью соответствует определению 1НФ. Но в реальности ради небольшого процента предельно редко и при том быстро выполняемых операций мы не станем «резать» атрибут и тем самым усложнять и замедлять все оставшиеся операции, которые оперируют значением этого атрибута как единым целым.

Так, например, дата вполне может храниться в поле типа `DATE` как единое целое, если у нас нет постоянно возникающей необходимости искать записи по принципу «все события, произошедшие в апреле и августе любого года» или «все события, произошедшие первого числа любого месяца».

Краткий вывод по первой нормальной форме:

- сделать «таблицу в таблице» вам не позволит СУБД;
- каждый многозначный атрибут обязательно нужно убрать путём вынесения его в новое отношение;
- каждый составной атрибут стоит тщательно исследовать, чтобы понять, *действительно* ли он составной (не атомарный), и, если это так, разделить его на несколько новых отдельных атрибутов.

Задание 3.3.а: существуют ли в базе данных «Банк»{408} схемы отношений, не находящиеся в первой нормальной форме? Если вы считаете, что «да», внесите в модель соответствующие правки для приведения таких схем отношений к соответствующей нормальной форме.

3.3.2. ВТОРАЯ НОРМАЛЬНАЯ ФОРМА

Если по какой-то причине вы пропустили или забыли содержимое раздела, посвящённого теории зависимостей{174}, сейчас — самое время освежить в памяти соответствующий материал (особенно стоит перечитать определения полной{183} и частичной{184} функциональных зависимостей).

{Здесь приведена т.н. «слабая формулировка» определения 2НФ. Она часто используется в различных учебных пособиях, но далее будет показано, почему 2НФ стоит рассматривать в «сильной формулировке» её определения.}

Переменная отношения находится во **второй нормальной форме** (2НФ, 2NF[169]) тогда и только тогда, когда она находится в первой нормальной форме{241}, и каждый её неперичный атрибут{23} функционально полно{183} зависит от первичного ключа{39}.

Упрощённо: ни один атрибут, не входящий в состав первичного ключа, не должен функционально зависеть от части первичного ключа.

Из такой формулировки определения 2НФ следует, что если первичный ключ отношения является простым{40}, и отношение находится в 1НФ, то оно автоматически находится и во 2НФ. Казалось бы, всё очень хорошо и просто (достаточно отношению, находящемуся в 1НФ, добавить простой искусственный{42} первичный ключ), но существует более полное определение 2НФ.

{Здесь приведена т.н. «сильная формулировка» определения 2НФ.}

Переменная отношение находится во **второй нормальной форме** (2НФ, 2NF[170]) тогда и только тогда, когда она находится в первой нормальной форме{241}, и каждый её неключевой атрибут{23} функционально полно{183} зависит от любого потенциального ключа{37}.

Упрощённо: ни один атрибут, не входящий в состав потенциального ключа, не должен функционально зависеть от части какого бы то ни было из потенциальных ключей.

Для начала рассмотрим пример нарушения второй нормальной формы в слабой формулировке (см. рисунок 3.3.d).

Отношение **`group`** описывает учебную группу, и его атрибуты означают следующее:

- g_number — порядковый номер группы в рамках года поступления;
- g_start_year — год поступления (год, когда группа начала учёбу);
- g_years — годы обучения (через сколько лет после начала обучения группа должна закончить учёбу);
- g_head — идентификатор старосты группы (внешний ключ).

Поскольку в каждый год поступления может быть сформировано несколько групп, у данного отношения есть лишь один очевидный потенциальный ключ — комбинация {**`g_number`**, **`g_start_year`**}, который и был выбран в качестве первичного.

Атрибут **`g_years`** не может быть частью потенциального ключа (мы видим, что его комбинации с атрибутами **`g_number`** и **`g_start_year`** дублируются.

---

169 A relation schema R is in **2NF** if every nonprime attribute A in R is fully functionally dependent on the primary key of R. («Fundamentals of Database Systems», Ramez Elmasri, Shamkant Navathe)

170 A relation schema R is in **2NF** if every nonprime attribute A in R is not partially dependent on any key of R. («Fundamentals of Database Systems», Ramez Elmasri, Shamkant Navathe)

Почему мы не можем рассматривать атрибут **gr_head** (один и тот же студент не может быть старостой нескольких групп, т.е. значение этого поля будет уникальным) как ещё один потенциальный ключ? Потому, что тогда мы были бы обязаны включить для этого поля свойство **NOT NULL** и сразу бы получили аномалию вставки{162} — при формировании группы необходимо было бы сразу назначить старосту, что объективно не будет выполняться в реальности в большинстве случаев.

Рассмотрим имеющиеся в представленном на рисунке 3.3.d отношении зависимости:

- Полная функциональная зависимость {**g_number**, **g_start_year**} → {**g_head**} показывает, что идентификатор старосты зависит от первичного ключа целиком (действительно — чтобы узнать старосту группы, необходимо гарантированно идентифицировать группу).
- Частичная функциональная зависимость {~~**g_number**~~, **g_start_year**} → {**g_years**} показывает, что продолжительность обучения зависит только от года поступления, но никак не зависит от номера группы в рамках года поступления. При этом мы видим, что раньше обучение занимало пять лет (для поступивших по 1999-й год включительно), а потом оно стало занимать четыре года (для поступивших в 2000-м году и позднее).

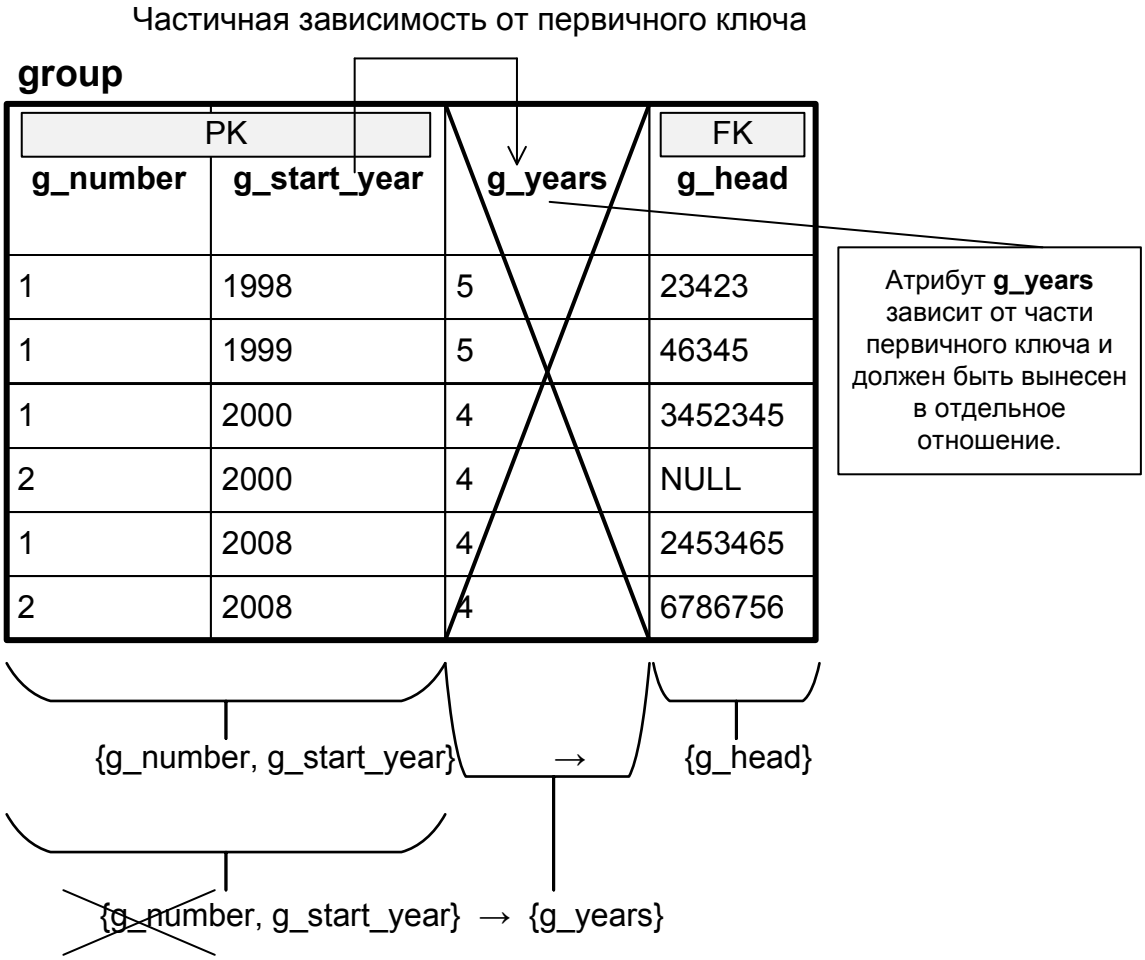


*Рисунок 3.3.d — **Нарушение второй нормальной формы (в «слабой формулировке»)***

Итак, мы видим, что существует неключевой атрибут (**`g_years`**), который зависит от части первичного ключа, т.е. «слабая формулировка» определения второй нормальной формы нарушена, и отношение group не находится во второй нормальной форме.

«Слабая формулировка» определения 2НФ получила распространение по той причине, что отношения с составными альтернативными ключами в реальной жизни встречаются нечасто. Т.е. если таких ключей в отношении нет, можно руководствоваться «слабой формулировкой» при нормализации ко 2НФ. Но если такие ключи есть (как будет показано далее) — необходимо руководствоваться «сильной формулировкой».

Прежде, чем обсуждать последствия нарушения второй нормальной формы и способы приведения к ней переменных отношений, рассмотрим нарушение второй нормальной формы в её «сильной формулировке». Для этого добавим в отношение group искусственный первичный ключ **`g_id`** (см. рисунок 3.3.е).

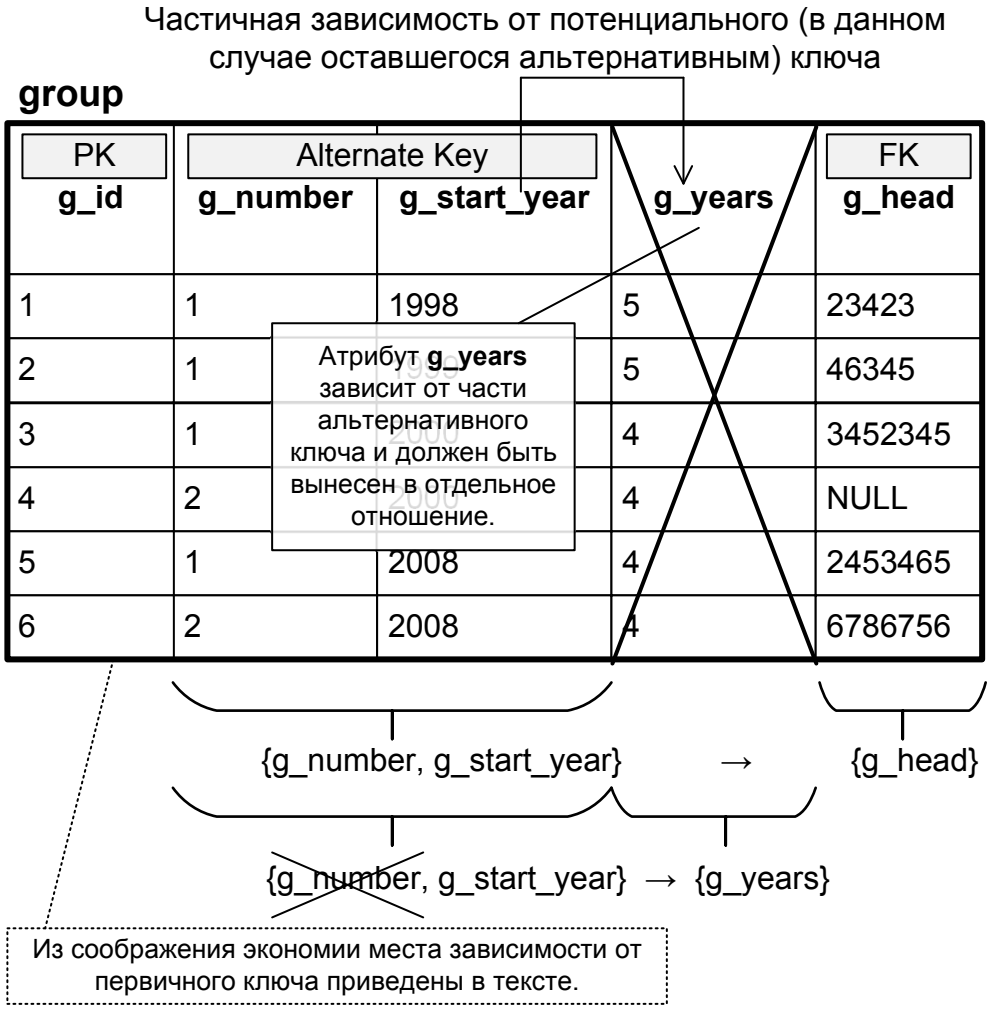


*Рисунок 3.3.е — **Нарушение второй нормальной формы (в «сильной формулировке»)***

Рассмотрим имеющиеся в представленном на рисунке 3.3.e отношении зависимости:

- Полная функциональная зависимость {**g_number**, **g_start_year**} → {**g_head**} не изменилась по сравнению с ситуацией, представленной на рисунке 3.3.d.
- Частичная функциональная зависимость {~~**g_number**~~, **g_start_year**} → {**g_years**} не изменилась по сравнению с ситуацией, представленной на рисунке 3.3.d.
- Появилось четыре новых функциональных зависимости:
  - {**g_id**} → {**g_number**}, т.к. уникальный идентификатор группы однозначно определяет её номер;
  - {**g_id**} → {**g_start_year**}, т.к. уникальный идентификатор группы однозначно определяет год начала обучения;
  - {**g_id**} → {**g_years**}, т.к. уникальный идентификатор группы однозначно определяет продолжительность обучения;
  - {**g_id**} → {**g_head**}, т.к. уникальный идентификатор группы однозначно определяет её старосту.

Итак, все неключевые атрибуты функционально полно зависят от первичного ключа, т.е. «слабая формулировка» определения второй нормальной формы выполняется, но «сильная формулировка» нарушена, т.к. есть атрибут **g_years** зависящий от части потенциального ключа {~~**g_number**~~, **g_start_year**}.

И теперь настало время показать, чем грозит нарушение второй нормальной формы. Допустим, что в предметной области существует правило (оно в полной мере выполнено на рисунках 3.3.d и 3.3.e): «для поступивших по 1999-й год включительно обучение занимает пять лет, а для поступивших в 2000-м году и позднее — четыре года».

Но что мешает нам нарушить это правило, добавив новые или изменив текущие данные так, чтобы (например) для группы, начавшей обучение в 1998-м году срок обучения составил 4 года (или 3, или 8, или любое иное число лет)? В текущем состоянии схемы базы данных — ничто.

Интуитивное желание включить контроль этого правила через триггер{350} может привести к обратной, но не менее опасной ситуации: что будет, если в будущем требования изменятся? Например, для поступивших «с 2015-го года и позднее» строк обучения снова должен стать пять лет.

Единственный вариант, гарантирующий как соблюдение взаимосвязи года поступления и срока обучения в настоящем, так и возможность корректировать это правило в будущем — создание отдельного отношения, как это показано на рисунке 3.3.f.

После выполнения нормализации единственный неключевой атрибут **g_head** отношения **group** функционально полно зависит от всех потенциальных ключей:

- {**g_id**} → {**g_head**}, т.е. идентификатор группы однозначно определяет её старосту;
- {**g_number**, **g_start_year**} → {**g_head**}, т.е. альтернативный способ гарантированно идентифицировать группу (альтернативный ключ{38}) однозначно определяет её старосту.

В отношении **education_length** единственный неключевой атрибут **el_years** функционально полно зависит от единственного потенциального ключа (который за неимением альтернатив и выбран первичным):

- {**el_start**} → {**el_years**}, т.е. год начала обучения однозначно идентифицирует продолжительность обучения.

Таким образом в обоих полученных отношениях нет неключевых атрибутов{23}, частично зависящих{184} от потенциальных ключей{37}, что полностью соответствует требованиям «сильной формулировки» определения второй нормальной формы.

И теперь, указав год начала обучения для новой группы, мы гарантированно верно указываем её продолжительность обучения.

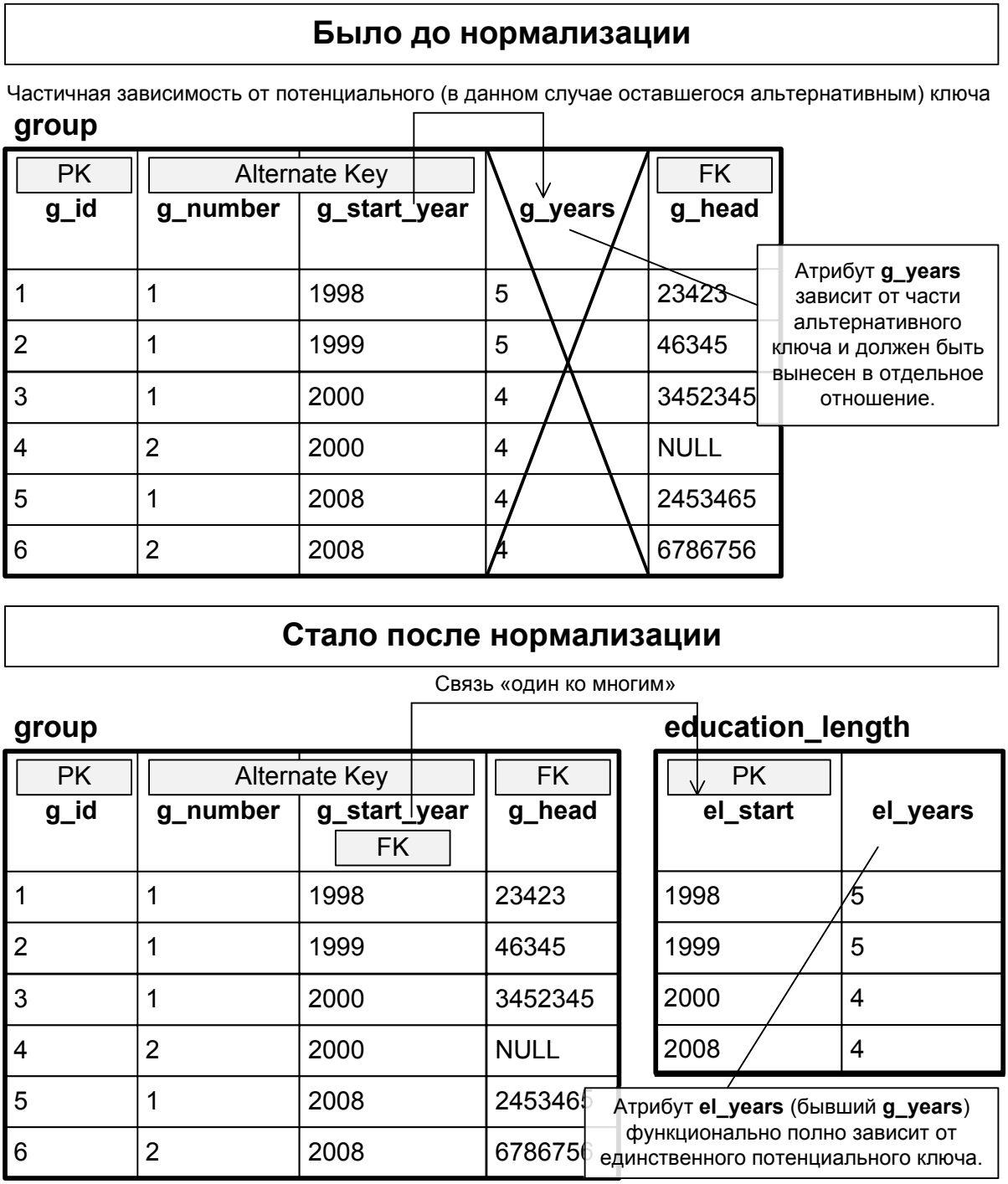


*Рисунок 3.3.f — **Приведение отношения ко второй нормальной форме***

Краткий вывод по второй нормальной форме:

- следует руководствоваться «сильной формулировкой» определения 2НФ и искать частичные зависимости неключевых атрибутов именно от потенциальных ключей, а не только лишь от первичного ключа;
- каждый атрибут, частично зависящий от потенциального ключа, стоит перенести в отдельное отношение, сделав первичным ключом этого нового отношения ту «часть потенциального ключа», от которой переносимый атрибут зависел до проведения нормализации.

Задание 3.3.b: Помимо возможности ошибочно указать продолжительность обучения в отношениях, представленных на рисунках 3.3.d и 3.3.e, там существует и вторая проблема: аномалии модификации данных. Рассмотрите внимательно эти отношения и определите, каким аномалиям они подвержены.

Задание 3.3.c: существуют ли в базе данных «Банк»{408} схемы отношений, не находящиеся во второй нормальной форме? Если вы считаете, что «да», внесите в модель соответствующие правки для приведения таких схем отношений к соответствующей нормальной форме.

3.3.3. ТРЕТЬЯ НОРМАЛЬНАЯ ФОРМА

Перед прочтением материала данной главы стоит повторить определения транзитивной{187} и нетривиальной{191} функциональных зависимостей).

{Здесь приведена т.н. «упрощённая формулировка» определения 3НФ.}

Переменная отношения находится в **третьей нормальной форме** (3НФ, 3NF[171]) тогда и только тогда, когда она находится во второй нормальной форме{246}, и каждый её неключевой атрибут{23} нетранзитивно{187} зависит от первичного ключа{39}.

Упрощённо: в отношении не должно быть атрибутов, не входящих в состав первичного ключа и при этом транзитивно зависящих от первичного ключа.

{Здесь приведена т.н. «каноническая формулировка» определения 3НФ.}

Переменная отношения R находится в **третьей нормальной форме** (3НФ, 3NF[172]) тогда и только тогда, когда она находится во второй нормальной форме{246}, и для каждой нетривиальной{191} функциональной зависимости {X} → A в этой переменной отношения выполняется хотя бы одно из двух условий:
а) {X} является суперключом{36} переменной отношения R;
б) A является ключевым атрибутом{23} переменной отношения R.

Упрощённо: если какой-то атрибут функционально зависит от множества других атрибутов, то должно выполняться хотя бы одно из двух условий:
а) это множество атрибутов уникально определяет любую запись;
б) этот зависимый атрибут входит в состав потенциального ключа.

В отличие от «слабой» и «сильной» формулировок определения 2НФ{246} эти определения лишь показывают два взгляда на одну и ту же ситуацию, причём «упрощённая» формулировка следует из «канонической». Покажем это.

- Поскольку (по определению{36}) любой суперключ уникально идентифицирует любую запись таблицы и (по определению{39}) первичный ключ уникально идентифицирует любую запись таблицы, то одному значению первичного ключа всегда соответствует строго одно значение любого суперключа, т.е. (по определению{180}) существует функциональная зависимость {первичный ключ} → {любой суперключ}. Также можно вспомнить, что первичный ключ сам по себе является частным случаем суперключа.
- Поскольку зависимость {первичный ключ} → {суперключ} → A является избыточной, приведённое в предыдущем пункте рассуждение позволяет в «канонической» формулировке понимать условие «а» как «{X} является первичным ключом» (что соответствует «упрощённой» формулировке).
- Условие «б» в «канонической» формулировке можно выразить через отрицание, т.е. вместо «A является ключевым атрибутом» сказать «A не является неключевым атрибутом» (что также соответствует «упрощённой формулировке»).

---

171 A relation schema R is in **3NF** if it satisfies 2NF and no nonprime attribute of R is transitively dependent on the primary key. («Fundamentals of Database Systems», Ramez Elmasri, Shamkant Navathe)

172 A relation schema R is in **3NF** if, whenever a nontrivial functional dependency X → A holds in R, either (a) X is a superkey of R, or (b) A is a prime attribute of R. («Fundamentals of Database Systems», Ramez Elmasri, Shamkant Navathe)

Рассмотрим пример нарушения третьей нормальной формы (рисунок 3.3.g). Здесь показаны два случая, соответствующие обоим формулировкам определениям 3НФ:

- по «упрощённой» формулировке определения получается, что существует транзитивная зависимость {g_id} → {g_faculty} → {g_dean} → {g_dean_dob}, т.е. от первичного ключа **`g_id`** зависит неключевой атрибут **`g_faculty`** (факультет, к которому относится группа), и уже от факультета зависит его декан (**`g_dean`**) и вся относящаяся к декану информация (в данном случае мы ограничились одним полем **`g_dean_dob`**, хранящим дату рождения декана);
- по «канонической» формулировке определения получается, что существует транзитивная зависимость {g_number, g_start_year} → {g_faculty} → {g_dean} → {g_dean_dob}, т.е. от альтернативного ключа {**`g_number`**, **`g_start_year`**} зависит неключевой атрибут **`g_faculty`** (факультет, к которому относится группа), и уже от факультета зависит его декан (**`g_dean`**) и вся относящаяся к декану информация (в данном случае мы ограничились одним полем **`g_dean_dob`**, хранящим дату рождения декана).

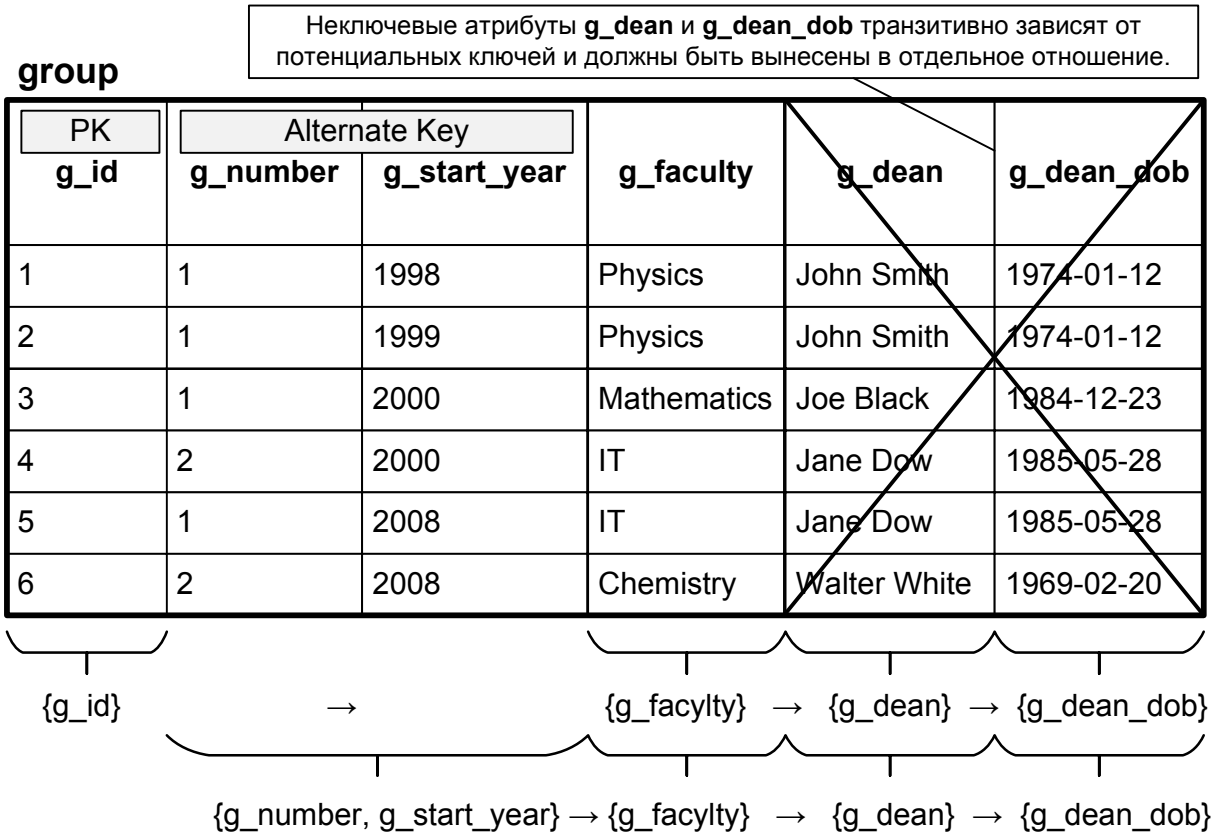


*Рисунок 3.3.g — **Нарушение третьей нормальной формы***

Нарушение третьей нормальной формы является нежелательным потому, что оно приводит к бессмысленному дублированию хранимой информации. Представьте, что на каждом факультете обучается по 100 групп: тогда мы будем для каждого факультета по 100 раз хранить имя декана и его дату рождения (а в совсем плохом случае и ещё несколько десятков значений данных, характеризующих декана).

Помимо бессмысленных затрат памяти, мы также получаем аномалию обновления{161}, т.к. при изменении какой-либо информации о декане её нужно будет обновить в записях всех групп, относящихся к данному факультету.

Часто можно услышать вопрос о том, в чём же принципиальное отличие нарушения 2НФ (обсуждая рисунок 3.3.e{248}, мы говорили, что можно ошибочно приписать не тот срок обучения некоторой группе) и нарушения 3НФ (ведь ничто не мешает, например, приписать некоторой группе не тот факультет).

Отличие состоит в том, что нарушение 2НФ позволяет отступить от некоего глобального правила предметной области (т.е. «**любая** группа, начавшая обучения в таком-то году, учится столько-то лет»), а нарушение 3НФ позволяет лишь ошибочно указать некое отдельное значение данных, не нарушая никакого глобального правила (да, мы можем ошибочно записать некую группу на не тот факультет, но этим мы не нарушим никакого правила вида «любая группа, соответствующая таким-то условиям, обучается на таком-то факультете», т.к. такого правила нет).

Рассмотрим приведение отношения к третьей нормальной форме (см. рисунок 3.3.h).

Ранее мы уже увидели, что атрибуты **g_dean** и **g_dean_dob** являются «лишними» в отношении **group** и должны быть перенесены в отдельное отношение, но как мы видим на рисунке 3.3.h, вместо одного нового отношения было создано два — **faculty** и **staff**.

У такого решения есть две причины.

Первая — тривиальная и очень практичная: логично предположить, что в базе данных будет храниться информация не только о деканах факультетов, но и об иных сотрудниках университета, лишь часть их которых является деканами. Нам было бы некуда поместить информацию о «не деканах», если бы мы не разделили описание факультетов и сотрудников.

Вторая — полу-интуитивная, но всё же заслуживающая упоминания. Если бы мы оставили информацию о деканах в отношении **faculty**, мы снова получили бы нарушение третьей нормальной формы, т.к. присутствовала бы транзитивная зависимость неключевого атрибута от потенциального ключа: {Факультет} → {Декан} → {Дата рождения декана}. А в отношении **staff** такой зависимости нет, т.к. дата рождения сотрудника не зависит от его имени.

Иными словами, атрибут, содержащий имя декана, играет разную роль в зависимости от того, в каком отношении он находится. В отношении **group** до нормализации значение этого атрибута отвечало на вопрос «Кто декан этого факультета, кто этот сотрудник?» и определяло собой всю информацию о сотруднике (включая дату рождения). В отношении **stuff** на вопрос «Кто этот сотрудник?» отвечает атрибут **s_id** (первичный ключ), и именно он определяет дату рождения и иные параметры сотрудника. А атрибут **s_name** отвечает только на вопрос «Каково имя этого сотрудника?» и на дату рождения никак не влияет: именно поэтому транзитивной зависимости {**s_id**} → {**s_name**} → {**s_dob**} нет, а есть лишь две функциональные зависимости {**s_id**} → {**s_name**} и {**s_id**} → {**s_dob**}, наличие которых не противоречит определению 3НФ.

Поскольку эта идея может быть неочевидной, но её понимание является важным для правильного приведения отношений к 3НФ, она показана графически на рисунке 3.3.i.

Было до нормализации

group

PK	Alternate Key				
g_id	**g_number**	**g_start_year**	**g_faculty**	**g_dean**	**g_dean_dob**
1	1	1998	Physics	John Smith	1974-01-12
2	1	1999	Physics	John Smith	1974-01-12
3	1	2000	Mathematics	Joe Black	1984-12-23
4	2	2000	IT	Jane Dow	1985-05-28
5	1	2008	IT	Jane Dow	1985-05-28
6	2	2008	Chemistry	Walter White	1969-02-20

Неключевые атрибуты **g_dean** и **g_dean_dob** транзитивно зависят от потенциальных ключей и должны быть вынесены в отдельное отношение.

Стало после нормализации

group

Связь «1-M»

PK	Alternate Key		FK
g_id	**g_number**	**g_start_year**	**g_faculty**
1	1	1998	1
2	1	1999	1
3	1	2000	2
4	2	2000	3
5	1	2008	3
6	2	2008	4

Теперь ни в одном из представленных здесь отношений нет транзитивной зависимости неключевых атрибутов от потенциальных ключей.

faculty

Связь «1-M»

staff

PK	Alt. Key	FK
f_id	**f_name**	**f_dean**
1	Physics	127
2	Mathematics	216
3	IT	678
4	Chemistry	993

PK		
s_id	**s_name**	**s_dob**
127	John Smith	1974-01-12
216	Joe Black	1984-12-23
678	Jane Dow	1985-05-28
993	Walter White	1969-02-20

*Рисунок 3.3.h — **Приведение отношения к третьей нормальной форме***

*Рисунок 3.3.i — **Изменение сути атрибута в зависимости от отношения, которому он принадлежит***

Краткий вывод по третьей нормальной форме:

- «упрощённая» и «каноническая» формулировки определения 3НФ эквивалентны, и можно руководствоваться любой из них;
- 3НФ не защищает отношение от возможности ошибочного указания каких-то локальных данных (например, ошибочно вписать номер паспорта человека), но т.к. отношение уже находится во 2НФ, будут соблюдаться «глобальные правила», распространяющиеся на множество записей;
- ярким признаком нарушения 3НФ является бессмысленное дублирование одних и тех же данных в множестве строк таблицы (тогда атрибуты, значения которых бессмысленно дублируются, являются первыми кандидатами на перемещение в новое отдельное отношение).

Задание 3.3.d: существуют ли в базе данных «Банк»{408} схемы отношений, не находящиеся в третьей нормальной форме? Если вы считаете, что «да», внесите в модель соответствующие правки для приведения таких схем отношений к соответствующей нормальной форме.

3.3.4. НОРМАЛЬНАЯ ФОРМА БОЙСА-КОДДА

Перед прочтением материала данной главы стоит повторить определения транзитивной{187} и нетривиальной{191} функциональных зависимостей).

{Здесь приведена т.н. «упрощённая формулировка» определения НФБК.}

Переменная отношения находится в **нормальной форме Бойса-Кодда** (НФБК, BCNF) тогда и только тогда, когда она находится во второй нормальной форме{246}, и каждый её атрибут нетранзитивно{187} зависит от первичного ключа{39}.

Упрощённо: в отношении не должно быть атрибутов, транзитивно зависящих от первичного ключа.

{Здесь приведена т.н. «каноническая формулировка» определения НФБК.}

Переменная отношения R находится в **нормальной форме Бойса-Кодда** (НФБК, BCNF[173]) тогда и только тогда, когда она находится во второй нормальной форме{246}, и в каждой нетривиальной{191} функциональной зависимости {X} → A в этой переменной отношения {X} является суперключом{36}.

Упрощённо: если какой-то атрибут функционально зависит от множества других атрибутов, это множество атрибутов уникально определяет любую запись.

Отличие НФБК от 3НФ состоит в том, что 3НФ допускает транзитивную зависимость ключевых атрибутов от суперключа, а НФБК утверждает, что ни для какого атрибута (в т.ч. ключевого) не должно существовать таких зависимостей.

Это отличие хорошо видно, если расположить определения рядом:

	3НФ	НФБК
Упрощённая формулировка	Переменная отношения находится в третьей нормальной форме тогда и только тогда, когда она находится во второй нормальной форме, и каждый её неключевой атрибут нетранзитивно зависит от первичного ключа.	Переменная отношения находится в нормальной форме Бойса-Кодда тогда и только тогда, когда она находится во второй нормальной форме, и каждый её ~~неключевой~~ атрибут нетранзитивно зависит от первичного ключа.
Каноническая формулировка	Переменная отношения R находится в третьей нормальной форме тогда и только тогда, когда она находится во второй нормальной форме, и для каждой нетривиальной функциональной зависимости {X} → A в этой переменной отношения выполняется хотя бы одно из двух условий: а) {X} является суперключом переменной отношения R; б) A является ключевым атрибутом переменной отношения R.	Переменная отношения R находится в нормальной форме Бойса-Кодда тогда и только тогда, когда она находится во второй нормальной форме, и для каждой нетривиальной функциональной зависимости {X} → A в этой переменной отношения выполняется ~~хотя бы одно из двух условий~~: а) {X} является суперключом переменной отношения R; ~~б) A является ключевым атрибутом переменной отношения R.~~

[173] A relation schema R is in **BCNF** if whenever a nontrivial functional dependency X → A holds in R, then X is a superkey of R. («Fundamentals of Database Systems», Ramez Elmasri, Shamkant Navathe)

Существует ещё две формулировки определения НФБК, которые могут оказаться более простыми для понимания и запоминания.

Переменная отношения находится в **нормальной форме Бойса-Кодда** (НФБК, BCNF[174]) тогда и только тогда, когда она находится в третьей нормальной форме{252}, и не содержит пересекающихся потенциальных ключей.

Упрощённо: отношение должно находиться в 3НФ, и в нём не должно быть потенциальных ключей{37}, имеющих общие атрибуты.

Переменная отношения находится в **нормальной форме Бойса-Кодда** (НФБК, BCNF[175]) тогда и только тогда, когда детерминанты{180} всех её функциональных зависимостей являются потенциальными ключами.

Упрощённо: в любой функциональной зависимости {X} → A множество {X} должно быть потенциальным ключом{37}.

Рассмотрим пример нарушения НФБК (см. рисунок 3.3.j). Отношение **`gift`** описывает подарочные наборы. При этом (в рамках предметной области) существуют договорённости:

- ни в каком наборе не может быть двух и более предметов одного типа;
- ни в каком наборе не может быть двух и более одноимённых предметов;
- для каждого типа предметов допустим только один вариант предмета.

Первая договорённость позволяет нам считать комбинацию значений полей {g_set, g_type} уникальной и объявить соответствующий составной первичный ключ.

Вторая договорённость позволяет нам также считать уникальной комбинацию значений полей {**`g_set`**, **`g_name`**}, но т.к. первичный ключ уже выбран, эта комбинация останется альтернативным ключом. И всё же атрибут **`g_name`** является ключевым (поэтому отношение **`gift`** не противоречит второй нормальной форме{246}: да, атрибут **`g_name`** зависит от части первичного ключа, но он сам является ключевым).

Третья договорённость даёт нам функциональную зависимость {**`g_type`**} → {**`g_name`**}, но мы видим (см. рисунок 3.3.j), что атрибут **`g_type`** не обладает свойством уникальности, т.е. детерминант функциональной зависимости не является потенциальным ключом, и потому отношение **`gift`** не находится в нормальной форме Бойса-Кодда.

Нарушение НФБК опасно тем же набором аномалий, что и нарушение второй нормальной формы: например, можно для набора подарков «Набор 2» перепутать наименования предметов и записать «игрушка — мармелад, сладости — медведь плюшевый», а потом в «Наборе 1» тоже что-то перепутать и записать «сладости — апельсины». Кроме того, что здесь перепутаны данные мы ещё получаем и нарушение только что упомянутой третьей договорённости, согласно которой для каждого типа предметов допустим лишь один вариант предмета, а у нас получилось, что «сладости» — это и «медведь плюшевый», и «апельсины».

---

174 A 3NF table that does not have multiple overlapping candidate keys is guaranteed to be in BCNF. «A Note on Relation Schemes Which Are in 3NF But Not in BCNF» (Vincent, M.W. and B. Srinivasan)

175 Relvar R is in BCNF if and only if every FD that holds in R is implied by some superkey. («The New Relational Database Dictionary», C.J. Date)

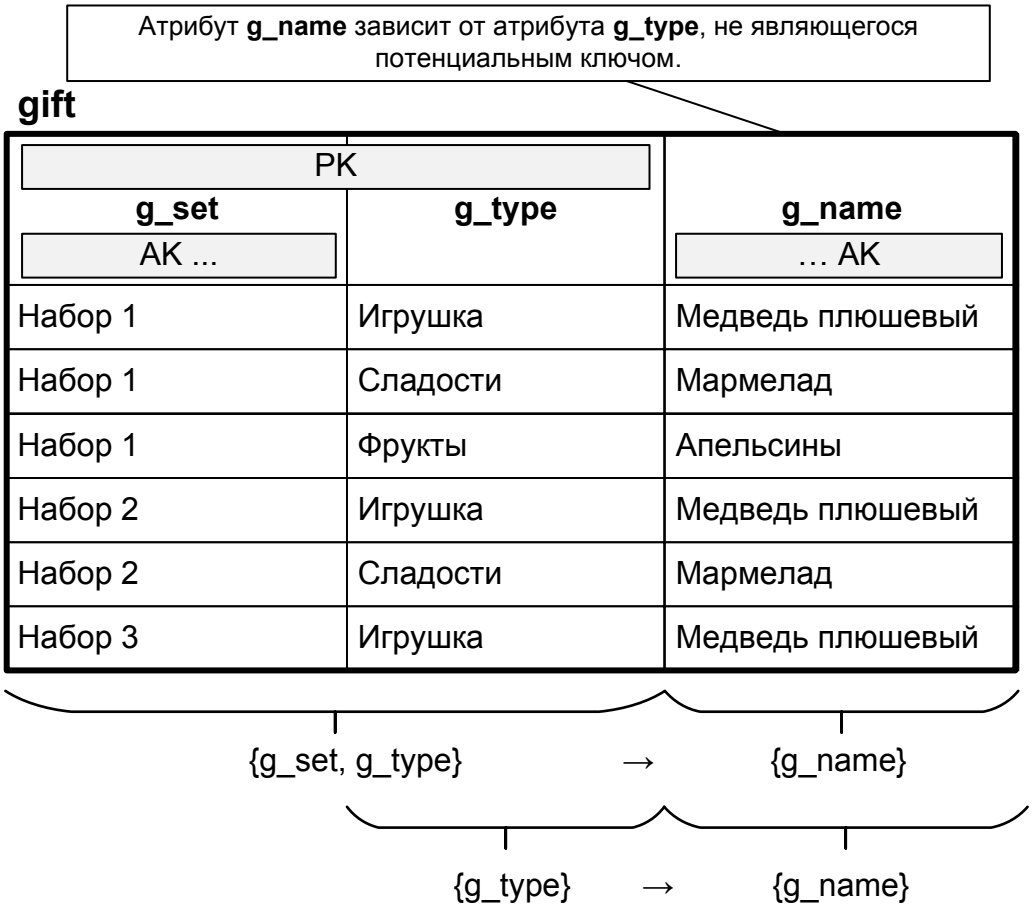


*Рисунок 3.3.j — **Нарушение нормальной формы Бойса-Кодда***

Чтобы привести это отношение к НФБК, его атрибуты необходимо распределить по новым отношениям.

Отметим, что здесь может быть сложность, т.к. не каждое распределение позволит потом получить исходное отношение через операцию объединения[176] (`JOIN`), но в общем случае достаточно «разрезать» исходное отношение по «проблемному» полю (являющемуся детерминантом функциональной зависимости и при этом не являющемуся потенциальным ключом; в нашем случае — это поле `g_type`).

Результат такого преобразования показан на рисунке рисунок 3.3.k.

[176] См. подробности в главе 15.5 книги «Fundamentals of Database Systems» (Ramez Elmasri, Shamkant Navathe).

*Рисунок 3.3.k — **Приведение отношения к нормальной форме Бойса-Кодда***

Краткий вывод по третьей нормальной форме:

все формулировки определения НФБК эквивалентны, и можно руководствоваться любой из них;

- как и 3НФ, НФБК не защищает отношение от возможности ошибочного указания каких-то локальных данных (например, можно ошибочно вписать название игрушки), но позволяет уменьшить дублирование данных и защищает отношение от нарушения «глобальных правил», распространяющихся на множество записей и несколько атрибутов;
- ярким признаком нарушения НФБК является наличие в отношении пересекающихся потенциальных ключей.

Задание 3.3.е: существуют ли в базе данных «Банк»[{408}] схемы отношений, не находящиеся нормальной форме Бойса-Кодда? Если вы считаете, что «да», внесите в модель соответствующие правки для приведения таких схем отношений к соответствующей нормальной форме.

3.3.5. ЧЕТВЁРТАЯ НОРМАЛЬНАЯ ФОРМА

Перед прочтением материала данной главы стоит повторить определения многозначной{193} зависимости и её подвидов — тривиальной{194} и нетривиальной{194}.

Переменная отношения R находится в **четвёртой нормальной форме** (4НФ, 4NF[177]) тогда и только тогда, когда для любой её нетривиальной многозначной зависимости X ↠ Y множество X является суперключом{36} R.

Упрощённо: отношение должно находиться в НФБК{258}, и в нём не должно быть нетривиальных{194} многозначных зависимостей.

Хорошим способом пояснения 4НФ является логика вывода упрощённого определения из строгого.

Почему отношение должно находиться в НФБК (см. упрощённую формулировку определения)?

Начнём с того, как связаны функциональная зависимость{180} и многозначная зависимость{193}. Фактически, функциональная зависимость — это многозначная зависимость, в которой количество значений функции ограничено единицей, т.е. A ↠ B эквивалентно A → B, когда мощность B равна единице[178].

Примем во внимание тот факт, что (см. строгую формулировку определения) многозначная зависимость X ↠ Y должна быть нетривиальной, потому функциональная зависимость X → Y также должна быть нетривиальной{191}, т.е. Y не может быть подмножеством X.

Это в полной мере соответствует определению НФБК{258}, которое для данного случая можно сформулировать так: «отношение R находится в НФБК тогда и только тогда, когда оно находится во второй нормальной форме, и для каждой не-тривиальной функциональной зависимости X → A в этом отношении X является суперключом отношения R».

Почему в отношении не должно быть нетривиальных многозначных зависимостей (см. упрощённую формулировку определения)?

Потому, что (см. строгую формулировку определения) в зависимости X ↠ Y множество X должно быть суперключом, т.е. в отношении не может находиться двух и более кортежей, значение совокупности атрибутов X в которых совпадает. А раз значение X может встретиться только один раз, то ему может соответствовать не более одного значения Y.

И это — как раз случай вырождения нетривиальной многозначной зависимости в тривиальную (см. рисунок 3.2.р{195}), т.е. нетривиальные многозначные зависимости в рамках 4НФ существовать не могут (что и сказано в упрощённой формулировке определения 4НФ).

Осталось привести графические примеры.

На рисунке 3.3.l представлен случай нарушения четвёртой нормальной формы. Так, если предметная область допускает подачу документов каждым абитуриентом на несколько факультетов, а на каждом факультете существует несколько вступительных экзаменов, то существуют зависимости {ue_applicant} ↠ {ue_faculty} и {ue_faculty} ↠ {ue_exam}, что можно записать как {ue_applicant} ↠ {ue_faculty} | {ue_exam}.

---

177 Relvar R is in **fourth normal form**, **4NF**, if and only if every MVD that holds in R is implied by some superkey of R — equivalently, if and only if for every nontrivial MVD X ↠ Y that holds in R, X is a superkey for R (in which case the MVD X ↠ Y effec-tively degenerates to the FD X → Y). («The New Relational Database Dictionary», C.J. Date)

178 См. подробное пояснение в разделе 13.2 книги «An Introduction to Database Systems (8th edition)» (C.J. Date).

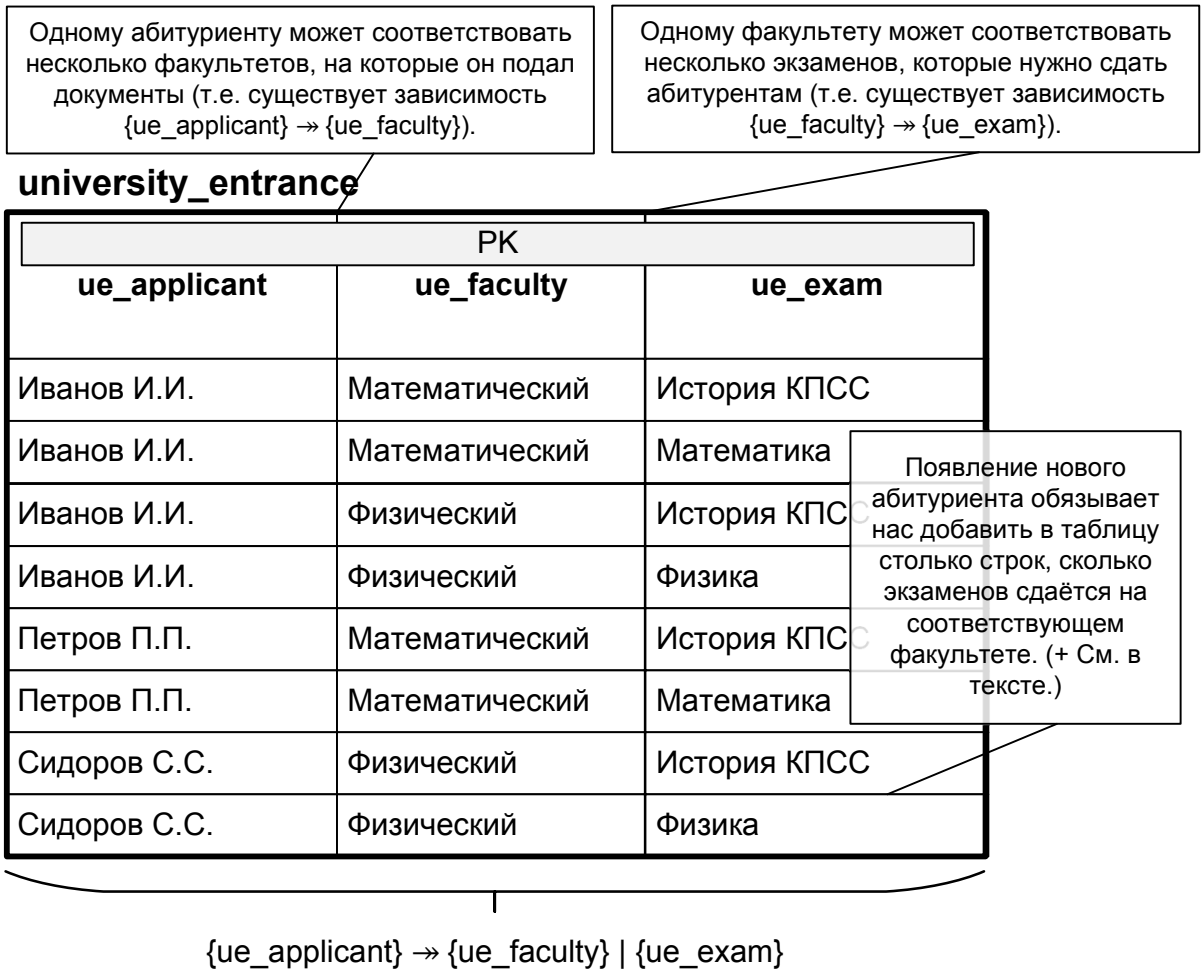


*Рисунок 3.3.l — **Нарушение четвёртой нормальной формы***

Отношение `university_entrance` находится в НФБК (у него нет неключевых атрибутов или пересекающихся потенциальных ключей), но оно не находится в четвёртой нормальной форме.

Нарушение четвёртой нормальной формы чревато возникновением серии аномалий операций с данными{161}:

- при появлении нового абитуриента запись о нём необходимо продублировать столько раз, сколько в совокупности будет экзаменов на всех факультетах, на которые он подал документы;
- при появлении нового экзамена на некотором факультете придётся добавить соответствующую запись столько раз, сколько абитуриентов хочет поступить на данный факультет;
- при добавлении нового факультета… мы не сможем его добавить, т.к. на него пока никто не подал документы, а поле `ue_applicant` входит в состав первичного ключа;
- аналогичные проблемы будут при обновлении и удалении данных, т.е. представленное на рисунке 3.3.l отношение является отличным примером реализации практически любой аномалии операций с данными.

Логика приведения отношения `university_entrance` к четвёртой нормальной форме показана на рисунке 3.3.m.

Было до нормализации

university_entrance

PK		
ue_applicant	**ue_faculty**	**ue_exam**
Иванов И.И.	Математический	История КПСС
Иванов И.И.	Математический	Математика
Иванов И.И.	Физический	История КПСС
Иванов И.И.	Физический	Физика
Петров П.П.	Математический	История КПСС
Петров П.П.	Математический	Математика
Сидоров С.С.	Физический	История КПСС
Сидоров С.С.	Физический	Физика

Одному абитуриенту может соответствовать несколько факультетов, на которые он подал документы (т.е. существует зависимость {ue_applicant} ↠ {ue_faculty}).

Одному факультету может соответствовать несколько экзаменов, которые нужно сдать абитурентам (т.е. существует зависимость {ue_faculty} ↠ {ue_exam}).

Стало после нормализации

application

PK	
a_applicant	**a_faculty**
Иванов И.И.	Математический
Иванов И.И.	Физический
Петров П.П.	Математический
Сидоров С.С.	Физический

exam

PK	
e_faculty	**e_exam**
Математический	История КПСС
Математический	Математика
Физический	История КПСС
Физический	Физика

*Рисунок 3.3.m — **Приведение отношения к четвёртой нормальной форме***

Отношения **`application`** и **`exam`** находятся в 4НФ, т.к. у них нет неключевых атрибутов, пересекающихся потенциальных ключей и «лишних» атрибутов, которые могли бы составить множество Z из определения нетривиальной многозначной зависимости[194].

Однако, такой результат приведения к 4НФ обладает очень неприятным недостатком: мы не можем провести между отношениями **`application`** и **`exam`** никакой связи, т.к. установка связи привела бы к миграции первичного ключа родительского отношения в дочернее, что дало бы нам исходное отношение **`university_entrance`**.

Потому с точки зрения практической реализации более выгодным является конечная схема, показанная на рисунке 3.3.n.

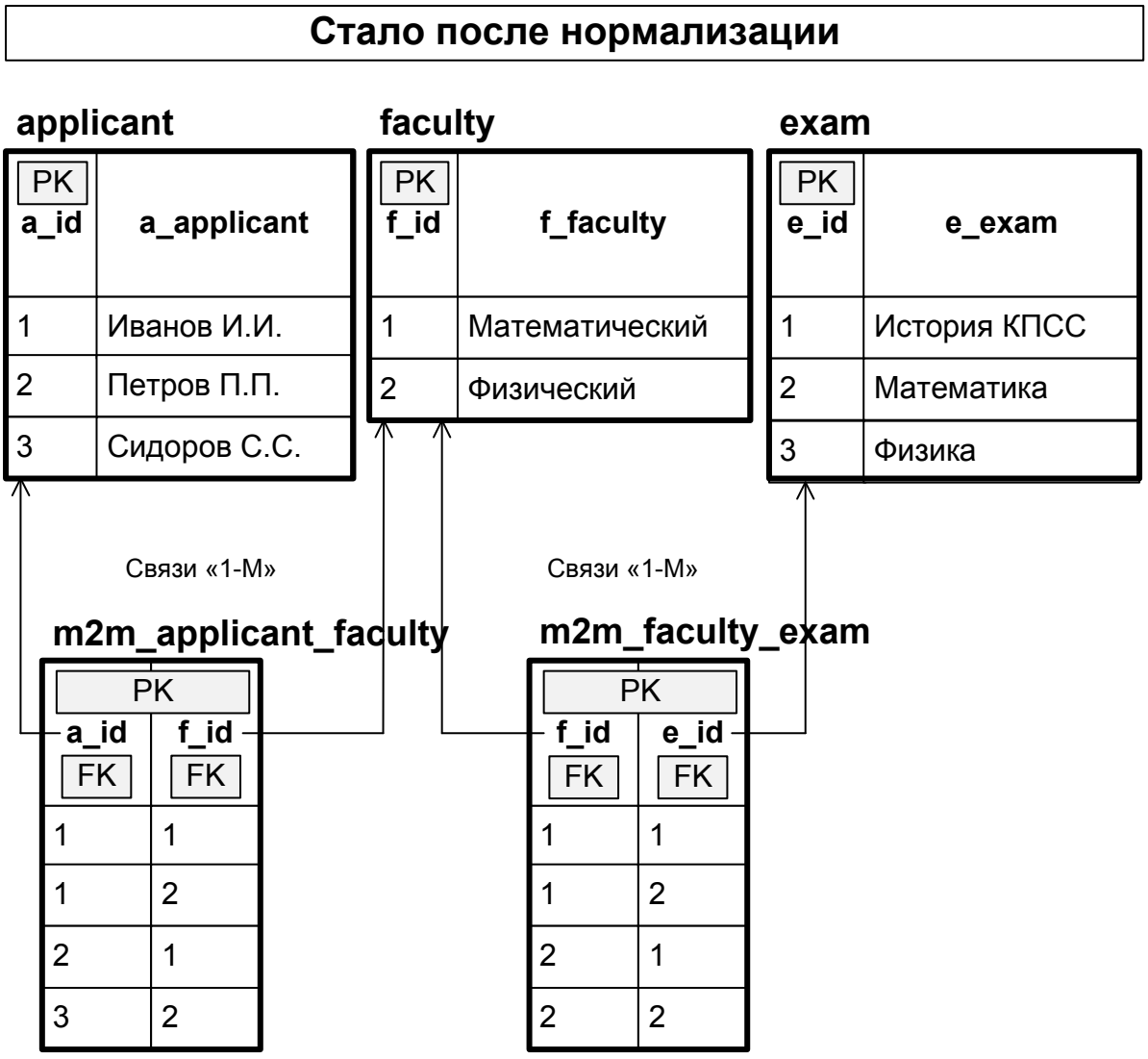


Рисунок 3.3.n — ***Более эффективное приведение отношения к четвёртой нормальной форме***

Пусть вариант, показанный на рисунке 3.3.n и содержит значительно больше отношений (т.е. при выполнении операций объединения СУБД придётся оперировать большим количеством объектов), он всё же позволяет задействовать механизм обеспечения ссылочной целостности{72}, и потому может считаться более предпочтительным, чем показанный на рисунке 3.3.m.

Краткий вывод по четвёртой нормальной форме:

- отношения, находящиеся в НФБК, но не находящиеся в 4НФ — большая редкость (на практике они появляются очень редко);
- как и НФБК, 4НФ не защищает отношение от возможности ошибочного указания каких-то локальных данных (например, можно ошибочно вписать название экзамена или приписать его «не тому» факультету), но позволяет уменьшить дублирование данных и защищает отношение от нарушения «глобальных правил», распространяющихся на множество записей и несколько атрибутов;
- ярким признаком нарушения 4НФ является наличие в отношении группы из трёх и более атрибутов, в которой атрибуты попарно зависят друг от друга, но при этом не зависят от значений остальных атрибутов.

Задание 3.3.f: существуют ли в базе данных «Банк»[{408}] схемы отношений, не находящиеся четвёртой нормальной форме? Если вы считаете, что «да», внесите в модель соответствующие правки для приведения таких схем отношений к соответствующей нормальной форме.

3.3.6. ПЯТАЯ НОРМАЛЬНАЯ ФОРМА

Перед прочтением материала данной главы стоит повторить определение зависимости соединения{196}.

Переменная отношения находится в **пятой нормальной форме** (5НФ[179], 5NF[180]) тогда и только тогда, когда она находится в 4НФ{263}, и для каждой существующей в ней нетривиальной зависимости соединения{196} JD(R_1, R_2, …, R_n) каждый набор атрибутов R_i является суперключом исходной переменной отношения.

Упрощённо: необходимо знать все потенциальные ключи отношения и все существующие в нём зависимости соединения, а также удостовериться, что каждая проекция, определяющая любую из зависимостей соединения, содержит потенциальный ключ.

Если вернуться к примерам наличия и отсутствия зависимости соединения (см. рисунки 3.2.r{198} и 3.2.s{200}), легко заметить что в реальной жизни крайне редко встречаются ограничения предметной области наподобие представленного ранее: «Если преподаватель ведёт некоторый предмет **П**, и на некотором факультете **Ф** преподаётся этот предмет **П**, и преподаватель ведёт хотя бы один любой предмет на этом факультете **Ф**, то он обязан вести на этом факультете **Ф** и предмет **П**».

Именно в силу редкости и сложности обнаружения в предметной области таких громоздких и нетривиальных правил говорят, что нормализация до 5НФ почти никогда не выполняется.

И всё же рассмотрим признаки нарушения 5НФ и способ её достижения. 5НФ нарушена, если (все условия должны быть выполнены одновременно):

- в отношении существуют зависимости соединения{196};
- хотя бы для одной такой зависимости существует проекция, не включающая в себя потенциальный ключ отношения.

Продолжим рассмотрение отношения, ранее представленного на рисунке 3.2.r{198}. Оно находится в 4НФ, т.к. здесь нет многозначных зависимостей{193}, однако с учётом только что процитированного правила про связь преподавателей, факультетов и предметов можно утверждать, что здесь есть зависимость соединения JD((`w_tutor`, `w_subject`), (`w_subject`, `w_faculty`), (`w_tutor`, `w_faculty`)).

Таким образом, первое условие нарушения 5НФ выполнено — зависимости соединения в этом отношении есть.

Выполнение второго условия нарушения 5НФ ещё более очевидно: ни одна из проекций (`w_tutor`, `w_subject`), (`w_subject`, `w_faculty`), (`w_tutor`, `w_faculty`) не содержит в себе потенциальных ключей (что легко проверить по представленным на рисунке 3.3.o данным).

---

179 5НФ иногда называют «нормальной формой проекции-соединения» (project-join normal form (PJNF)).

180 A relation schema R is in fifth normal form (5NF) (or project-join normal form (PJNF)) with respect to a set F of functional, multivalued, and join dependencies if, for every nontrivial join dependency JD(R_1, R_2, ..., R_n) in F^+ (that is, implied by F), every R_i is a superkey of R. («Fundamentals of Database Systems», Ramez Elmasri, Shamkant Navathe)

workload

PK		
w_tutor	**w_subject**	**w_faculty**
Иванов И.И.	Математика	Точных наук
Иванов И.И.	Информатика	Естествознания
Петров П.П.	Математика	Естествознания
Сидоров С.С.	Информатика	Кибернетики
Петров П.П.	Физика	Точных наук
Петров П.П.	Математика	Точных наук
Иванов И.И.	Математика	Естествознания

Присутствует зависимость соединения:
JD((w_tutor, w_subject),
(w_subject, w_faculty),
(w_tutor, w_faculty))

Ни одна из проекций (w_tutor, w_subject), (w_subject, w_faculty), (w_tutor, w_faculty) не содержит в себе потенциальных ключей.

*Рисунок 3.3.о — **Нарушение пятой нормальной формы***

Нарушение 5НФ, наличие которого мы только что доказали, говорит о том, что данное отношение можно декомпозировать{180} без потерь на несколько отдельных отношений. Результат такой декомпозиции представлен на рисунке 3.3.р.

Было до нормализации

workload

PK		
w_tutor	**w_subject**	**w_faculty**
Иванов И.И.	Математика	Точных наук
Иванов И.И.	Информатика	Естествознания
Петров П.П.	Математика	Естествознания
Сидоров С.С.	Информатика	Кибернетики
Петров П.П.	Физика	Точных наук
Петров П.П.	Математика	Точных наук
Иванов И.И.	Математика	Естествознания

Присутствует зависимость соединения:
JD((w_tutor, w_subject),
(w_subject, w_faculty),
(w_tutor, w_faculty))

Ни одна из проекций (w_tutor, w_subject), (w_subject, w_faculty), (w_tutor, w_faculty) не содержит в себе потенциальных ключей.

Стало после нормализации

workload_ts

PK	
w_tutor	**w_subject**
Иванов И.И.	Математика
Иванов И.И.	Информатика
Петров П.П.	Математика
Сидоров С.С.	Информатика
Петров П.П.	Физика

workload_sf

PK	
w_subject	**w_faculty**
Математика	Точных наук
Информатика	Естествознания
Математика	Естествознания
Информатика	Кибернетики
Физика	Точных наук

workload_tf

PK	
w_tutor	**w_faculty**
Иванов И.И.	Точных наук
Иванов И.И.	Естествознания
Петров П.П.	Естествознания
Сидоров С.С.	Кибернетики
Петров П.П.	Точных наук

*Рисунок 3.3.p — **Приведение отношения к пятой нормальной форме***

Как и в случае с 4НФ[263] мы получили вариант, корректный с точки зрения реляционной теории, но крайне неудобный на практике: мы не можем провести между полученными в процессе нормализации отношениями никакой связи, т.к. установка связи привела бы к миграции первичного ключа родительского отношения в дочернее, что дало бы нам исходное отношение **`workload`**.

Потому с точки зрения практической реализации более выгодным является конечная схема, показанная на рисунке 3.3.q.

Рисунок 3.3.q — ***Более эффективное приведение отношения к пятой нормальной форме***

Вариант, показанный на рисунке 3.3.q содержит значительно больше отношений (т.е. при выполнении операций объединения СУБД придётся оперировать большим количеством объектов), но он позволяет задействовать механизм обеспечения ссылочной целостности{72}, и потому может считаться более предпочтительным, чем показанный на рисунке 3.3.p. Также данный вариант позволяет более эффективно использовать триггеры{350} для контроля выполнения бизнес-правила о связи между преподавателями, предметами и факультетами.

В завершении этой главы отметим, что если бы в предметной области не существовало правила о связи между преподавателями, предметами и факультетами, в отношении `workload` не было бы зависимости соединения, и оно уже изначально находилось бы в 5НФ.

Краткий вывод по пятой нормальной форме:

- отношения, находящиеся в 4НФ, но не находящиеся в 5НФ — большая редкость (на практике они почти никогда не встречаются);
- как и 4НФ, 5НФ не защищает отношение от возможности ошибочного указания каких-то локальных данных (например, можно ошибочно вписать название факультета или приписать его «не тому» предмету), но позволяет уменьшить дублирование данных и защищает отношение от нарушения «глобальных правил», распространяющихся на множество записей и несколько атрибутов;
- признаком нарушения 5НФ является наличие в отношении группы из трёх и более атрибутов, в которой все входящие в эту группу атрибуты находятся во взаимосвязи друг от друга;
- 5НФ является «финальной» нормальной формой, т.е. находящееся в 5НФ отношение уже невозможно без потерь декомпозировать дальше (за исключением случаев хронологических отношений — см. 6НФ{273}).

Задание 3.3.g: существуют ли в базе данных «Банк»{408} схемы отношений, не находящиеся пятой нормальной форме? Если вы считаете, что «да», внесите в модель соответствующие правки для приведения таких схем отношений к соответствующей нормальной форме.

3.3.7. ДОМЕННО-КЛЮЧЕВАЯ НОРМАЛЬНАЯ ФОРМА

Перед прочтением материала данной главы стоит повторить определения доменной{201} и ключевой{201} зависимостей.

Переменная отношения находится в **доменно-ключевой нормальной форме** (ДКНФ, DKNF[181, 182]) тогда и только тогда, когда все ограничения и зависимости, существующие в этой переменной отношения, являются следствиями ограничений доменов{22} и ключей{35}.

Упрощённо: все применимые к отношению правила напрямую следуют из свойств его полей (и групп полей), и не существует никаких «скрытых» правил, вытекающих из чего бы то ни было другого.

ДКНФ является одновременно одной из самых простых и самых сложных. Её простота следует напрямую из определения (и как было показано ранее, из определений доменной{201} и ключевой{201} зависимостей).

Действительно, как кажется, нет ничего проще, чем обеспечить уникальность ключей отношения и контроль принадлежности любого значения любого атрибута соответствующему домену — современные СУБД предоставляют широчайший спектр технических возможностей…

Но это лишь кажется.

Напомним, что в определении ключевой зависимости{201} речь идёт о предметной области, а **не** о технической реализации. Таким образом, для любой таблицы с суррогатным{42} первичным ключом остаётся открытым вопрос о поиске потенциальных ключей и обеспечении гарантированной уникальности их значений для каждой записи (что с одной стороны ставит под сомнение смысл применения суррогатного ключа, а с другой стороны бывает само по себе непросто в реальных ситуациях, где потенциальные ключи{37} могут не всегда быть очевидными).

Что касается доменной зависимости{201}, то она тоже проста лишь в теории. Да, легко определить набор значений для дней недели или диапазон значений для роста человека. Но попробуйте решить те же задачи для таких доменов как «имя», «адрес», «описание товара» и им подобных — мы всегда будем вынуждены искать компромисс между вариантами «создать слишком сильное ограничение, не пропускающее часть допустимых значений» и «создать слишком слабое ограничение, пропускающее часть недопустимых значений».

И это — лишь «верхушка айсберга». По определению ДКНФ, в переменной отношения не должно быть никаких «скрытых правил». Но что, если в предметной области существуют (например) такие правила:

- новая зарплата сотрудника не может быть меньше старой;
- отпуск не должен начинаться в пятницу;
- зелёные птички должны получать на треть больше корма, чем жёлтые;
- запрещено выдавать скидку на продажу автомобилей белого цвета;
- должно быть не менее трёх пользователей с правами администратора.

---

181 A relation schema is said to be in **DKNF** if all constraints and dependencies that should hold on the valid relation states can be enforced simply by enforcing the domain constraints and key constraints on the relation. The idea behind **DKNF** is to specify (theoretically, at least) the *ultimate normal form* that takes into account all possible types of dependencies and constraints. («Fundamentals of Database Systems», Ramez Elmasri, Shamkant Navathe)

182 A 1NF relation schema is in **DK/NF** if every constraint can be inferred by simply knowing the DDs (domain dependencies) and the KDs (key dependencies). Putting it another way: a 1NF relation schema is in **DK/NF** if by enforcing the DDs and KDs, every constraint of the schema is automatically enforced. («A Normal Form for Relational Databases That Is Based on Domains and Keys», Ronald Fagin)

Все эти правила (и тысячи им подобных) никак не следуют ни из уникальности ключей, ни из набора значений какого бы то ни было поля.

Именно поэтому ДКНФ с одной стороны представляет собой очень красивое математическое решение[183], и действительно переменная отношения, находящаяся в ДКНФ, автоматически находится во всех предшествующих{280} ей нормальных формах (при допущении, что мощность доменов бесконечна).

Но с другой стороны ДКНФ настолько сложно достижима на практике, что её существование представляет скорее научный, чем практический интерес — и проблема не в самой ДКНФ, а в сложности предметных областей и таких их правил, которые самим фактом своего существования противоречат определению ДКНФ.

Если же отношение привести к ДКНФ «насильственно» (т.е. выполнить нормализацию ради нормализации), скорее всего, придётся попрощаться с возможностью выполнить ключевые требования{12} к базе данных.

Существует ещё один интересный факт: ДКНФ — единственная нормальная форма, которая не входит в иерархию нормальных форм, но подробнее об этом будет сказано в соответствующем разделе{280}.

И всё же, несмотря на всю необычность ДКНФ, рассмотрим небольшой пример[184]. Отношение **`employee`** (см. рисунок 3.3.r) описывает сотрудников и содержит два поля — **`name`** и **`role`**. Поле role может хранить данные как о роли в смысле «руководитель», «помощник руководителя» и т.д. (фактически, о должности сотрудника), так и о роли в смысле «программист», «архитектор», «тестировщик» и т.д. (фактически, о функции сотрудника), причём известно, что у сотрудника может быть только одно значение свойства «должность», но при этом может быть сколь угодно много значений свойства «функция».

Если только что вы подумали, что это — какое-то безумие, и что в реальности «должность» и «функция» находились бы в разных полях (и даже разных отношениях), вы совершенно правы. Но подчеркнём, что это — всего лишь наглядный пример приведения отношения к ДКНФ, и с чисто формальной точки зрения здесь нет никаких нестыковок.

Это отношение находится в 5НФ{268} (все его возможные проекции будут содержать суперключ исходного отношения, хотя тут всё даже проще: все проекции этого отношения будут являться исходным отношением), однако ДКНФ здесь нарушена: в предметной области есть ограничение (у сотрудника только одна должность), не вытекающее напрямую ни из доменной{201}, ни из ключевой{201} зависимостей.

К слову, формально ничто не мешает выполнить вставку в такое отношение записи о том же самом сотруднике, но с другим значением свойства «должность», т.е. нарушить требование предметной области.

Чтобы привести данное отношение к ДКНФ, необходимо разбить его на два отдельных отношения, в первом из которых будет храниться информация о должности сотрудника, а во втором — о его функциях.

Обратите внимание: здесь даже не идёт речь о проекциях исходного отношения: у новых отношений появились свои отдельные атрибуты, которых не было в исходном отношении.

---

183 Рональд Фагин в своей статье «A Normal Form for Relational Databases That Is Based on Domains and Keys» порядка двадцати страниц отводит на формулы и доказательства.

184 Данный пример заимствован у Рональда Фагина из его статьи «A Normal Form for Relational Databases That Is Based on Domains and Keys».

Было до нормализации

employee

PK	
e_name	**e_role**
Иванов И.И.	**Координатор**
Иванов И.И.	Программист
Иванов И.И.	Аналитик
Петрова П.П.	Руководитель
Петрова П.П.	Программист
Петрова П.П.	Архитектор
Иванов И.И.	**Руководитель**

Возможность вставки таких данных нарушает требование предметной области «у каждого сотрудника должна быть только одна должность», и само это правило никак не следует из ограничений ключей и/или доменов.

Стало после нормализации

employee_position

PK	
ep_name	**ep_position**
Иванов И.И.	Координатор
Петрова П.П.	Руководитель

employee_function

PK	
ef_name FK	**ef_function**
Иванов И.И.	Программист
Иванов И.И.	Аналитик
Петрова П.П.	Программист
Петрова П.П.	Архитектор

Теперь требование предметной области «у каждого сотрудника должна быть только одна должность», следует из ограничения ключей.

*Рисунок 3.3.r — **Приведение отношения к доменно-ключевой нормальной форме***

Краткий вывод по доменно-ключевой нормальной форме:

- не каждое отношение может быть приведено в ДКНФ;
- если отношение находится в ДКНФ, оно находится и в 5НФ;
- признаком нарушения ДКНФ (или даже невозможности приведения отношения к ДКНФ) является наличие в предметной области таких правил, которые не следуют напрямую из множеств значений атрибутов отношения и/или свойства уникальности ключей;
- приведение к ДКНФ, как правило, выполняется не декомпозицией отношения, а созданием новых отношений с новыми атрибутами (такими, каких не было в исходном отношении).

Задание 3.3.h: существуют ли в базе данных «Банк»{408} схемы отношений, не находящиеся доменно-ключевой нормальной форме? Если вы считаете, что «да», внесите в модель соответствующие правки для приведения таких схем отношений к соответствующей нормальной форме.

3.3.8. ШЕСТАЯ НОРМАЛЬНАЯ ФОРМА

Перед прочтением материала данной главы стоит повторить определение обобщённой зависимости соединения{204}.

Переменная отношения находится в **шестой нормальной форме** (6НФ, 6NF[185]) тогда и только тогда, когда она не допускает в принципе никакой декомпозиции без потерь{180}, т.е. любые имеющиеся в ней зависимости соединения{196}, {204} являются тривиальными{196}, {204}.

Упрощённо: при декомпозиции такой переменной отношения хотя бы одна из получившихся проекций всегда будет эквивалентна исходной переменной отношения.

На рисунке 3.3.s показана та же ситуация, что являлась иллюстрацией обобщённой зависимости соединения{204}, однако теперь мы посмотрим на представленные отношения с несколько иной точки зрения.

Исходное отношение `education_path` содержит информацию о том, в какой период времени на каком факультете и в какой группе обучался студент. Однако совершенно очевидно[186], что факультет и номер группы могут меняться независимо друг от друга.

Это приводит к очень нетривиальным операциям по поддержанию отношения в консистентном состоянии. Попробуйте сами определить, как изменится отношение, если студент с идентификатором 13452:

- в феврале 2018-го года обучался в группе 3 (но сейчас это просто забыли указать и хотят исправить ситуацию);
- уходил в августе 2018-го года на месяц в академический отпуск, в связи с чем продолжал числиться на химическом факультете, но не входил ни в одну группу;
- начиная с 2019-го года каждый месяц менял группу, оставаясь на одном и том же факультете.

Часть этих операций будет относительно простой и («всего лишь») приведёт к бессмысленному дублированию данных. Но есть здесь и операции, которые потребуют весьма непростого изменения отношения.

Ситуация намного упростилась бы, если бы мы могли хранить информацию о периодах пребывания студента на некотором факультете и в некоторой группе независимо.

Именно это и достигается декомпозицией отношения `education_path` на две проекции P_1 и P_2, причём никакая информация в результате такой декомпозиции не теряется: мы по-прежнему всегда можем выяснить для любого момента времени, на каком факультете и в какой группе обучался студент с идентификатором 13452.

Дальнейшая же декомпозиция проекций P_1 и P_2 невозможна, т.к. мы не можем «рвать на части» их первичные ключи (мы потеряем информацию «о каком студенте в какой период времени идёт речь») или «отрывать» неключевое поле от первичного ключа (сами по себе название факультета или номер группы без привязки к студенту и периоду времени бессмысленны), потому у нас для обеих проекций остаётся только такой вариант декомпозиции: отдельно сохранить отношение со значениями первичного ключа и отдельно… сохранить исходное отношение.

---

185 Relvar R is in sixth normal form, 6NF, if and only if it can't be nonloss decomposed at all, other than trivially — i.e., if and only if the only JDs to which it's subject are trivial ones. Equivalently, relvar R is in 6NF if and only if it's in 5NF, is of degree n, and has no key of degree less than n-1. («The New Relational Database Dictionary», C.J. Date)

186 Да, такие слова часто бесят ☺, но… я вас уверяю, уже это — ТОЧНО совершенно очевидно.

Таким образом, имеющиеся в проекциях P_1 и P_2 (обобщённые) зависимости соединения являются тривиальными, т.е. отношения P_1 и P_2 не содержат нетривиальных (обобщённых) зависимостей соединения, т.е. они никак не могут быть декомпозированы без потерь, т.е. они находятся в шестой нормальной форме.

Было до нормализации

education_path

PK			
ep_student	**ep_period**	**ep_faculty**	**ep_group**
13452	01.01.2018-31.05.2018	Химический	1
13452	01.03.2018-31.05.2018	Химический	5
13452	01.06.2018-01.12.2018	Физический	5

Факультет и номер группы могут независимо меняться в рамках любого интервала времени, что очевидно приводит к простой мысли: можно отдельно хранить интервалы времени для факультета и номера группы, т.е. декомпозировать отношение, приведя его проекции к более высокой нормальной форме.

Стало после нормализации

P_1 (ep_student, ep_period, ep_faculty)

PK		
ep_student	**ep_period**	**ep_faculty**
13452	01.01.2018-31.05.2018	Химический
13452	01.06.2018-01.12.2018	Физический

P_2 (ep_student, ep_period, ep_group)

PK		
ep_student	**ep_period**	**ep_group**
13452	01.01.2018-31.05.2018	1
13452	01.03.2018-01.12.2018	5

Теперь оба полученных отношения хранят (каждое в отдельности) информацию о том, в какой период времени студент обучался на каком факультете, и в какой период времени студент был в какой группе.

*Рисунок 3.3.s — **Приведение отношения к шестой нормальной форме***

Обратите внимание на вторую часть определения[185], где сказано, что переменная отношения арности *n* находится в 6НФ, если она находится в 5НФ и не содержит ключей арности меньшей, чем *n-1*.

Проиллюстрируем это на примере всё тех же отношений, представленных на рисунке 3.3.s.

Отношение **`education_path`** имеет арность, равную 4 (в этом отношении 4 атрибута) и находится в 5НФ, т.к. все его допустимые проекции содержат его же первичный ключ.

Однако, первичный ключ в том отношении имеет арность 2, т.е. 2 < (4-1), а по определению 6НФ таких ключей в отношении быть не должно.

В свою очередь проекции **`P1`** и **`P2`** имеют арность, равную 3, а их ключи — арность, равную двум. Итого условие 2 < (3-1) не выполняется, т.е. в этих отношениях нет ключей, арность которых меньше «арности отношения -1». По этому признаку они также находятся в 6НФ.

Краткий вывод по шестой нормальной форме:

- 6НФ актуальна только для хронологических отношений, для всех остальных финальной нормальной формной является 5НФ{268};
- 6НФ позволяет значительно упростить операции с отношением, вызванные необходимостью отдельно управлять значениями атрибутов, связанных с интервалами времени, но не связанных друг с другом;
- признаком нарушения 6НФ является наличие в отношении двух и более атрибутов, не связанных друг с другом, но связанных с одним и тем же хронологическим атрибутом;
- 6НФ является «совсем финальной» ☺ нормальной формой, т.е. любое находящееся в 6НФ отношение уже никак невозможно без потерь декомпозировать дальше.

Задание 3.3.i: существуют ли в базе данных «Банк»{408} схемы отношений, не находящиеся шестой нормальной форме? Если вы считаете, что «да», внесите в модель соответствующие правки для приведения таких схем отношений к соответствующей нормальной форме.

3.3.9. ИЕРАРХИЯ НОРМАЛЬНЫХ ФОРМ И НЕКАНОНИЧЕСКИЕ НОРМАЛЬНЫЕ ФОРМЫ

По аналогии с тем, как ранее был представлен краткий список всех основных зависимостей{208}, приведём такой же список всех основных нормальных форм.

Пожалуйста, используйте приведённые ниже определения только как способ быстрого запоминания сути той или иной нормальной формы. Полноценные строгие определения можно быстро найти, проследовав по ссылке, представленной рядом с англоязычным вариантом каждого термина.

Переменная отношения находится в **нулевой нормальной форме** (0НФ, 0NF{224}), когда к ней не предъявляется никаких особых требований, и допускаются любые нарушения любых нормальных форм.

Переменная отношения находится в **первой нормальной форме** (1НФ, 1NF{241}), когда каждый её атрибут атомарен (т.е. СУБД не должна оперировать никакой отдельной частью атрибута).

Переменная отношение находится во **второй нормальной форме** (2НФ, 2NF{246}), когда она находится в 1НФ, и ни один атрибут, не входящий в состав первичного ключа, не зависит функционально от части первичного ключа.

Переменная отношения находится в **третьей нормальной форме** (3НФ, 3NF{252}), когда она находится во 2НФ и не содержит атрибутов, не входящих в состав первичного ключа и при этом транзитивно зависящих от первичного ключа.

Переменная отношения находится в **нормальной форме Бойса-Кодда** (НФБК, BCNF{258}), когда она находится во 2НФ и не содержит атрибутов, транзитивно зависящих от первичного ключа.

Для запоминания разницы между 3НФ и НФБК:

3НФ	НФБК
Переменная отношения находится в 3НФ, когда она находится во 2НФ, и каждый её **неключевой** атрибут нетранзитивно зависит от первичного ключа.	Переменная отношения находится в НФБК, когда она находится во второй нормальной форме, и каждый её ~~неключевой~~ атрибут нетранзитивно зависит от первичного ключа.

Переменная отношения находится в **четвёртой нормальной форме** (4НФ, 4NF{263}), когда она находится в НФБК и не содержит нетривиальных многозначных зависимостей.

Переменная отношения находится в **пятой нормальной форме** (5НФ, 5NF{268}), когда она находится в 4НФ и каждая её проекция, определяющая любую из зависимостей соединения, содержит потенциальный ключ.

Переменная отношения находится в **доменно-ключевой нормальной форме** (ДКНФ, DKNF{273}), когда все применимые к ней правила напрямую следуют из свойств её полей (и групп полей), и не существует никаких «скрытых» правил, вытекающих из чего бы то ни было другого.

Переменная отношения находится в **шестой нормальной форме** (6НФ, 6NF{277}), когда она не допускает в принципе никакой декомпозиции без потерь, т.е. при декомпозиции такой переменной отношения хотя бы одна из получившихся проекций всегда будет эквивалентна исходной переменной отношения.

Схематично всю иерархию нормальных форм можно представить своеобразной пирамидой (см. рисунок 3.3.t), в которой рассмотренные нами ранее нормальные формы отмечены **жирным шрифтом**.

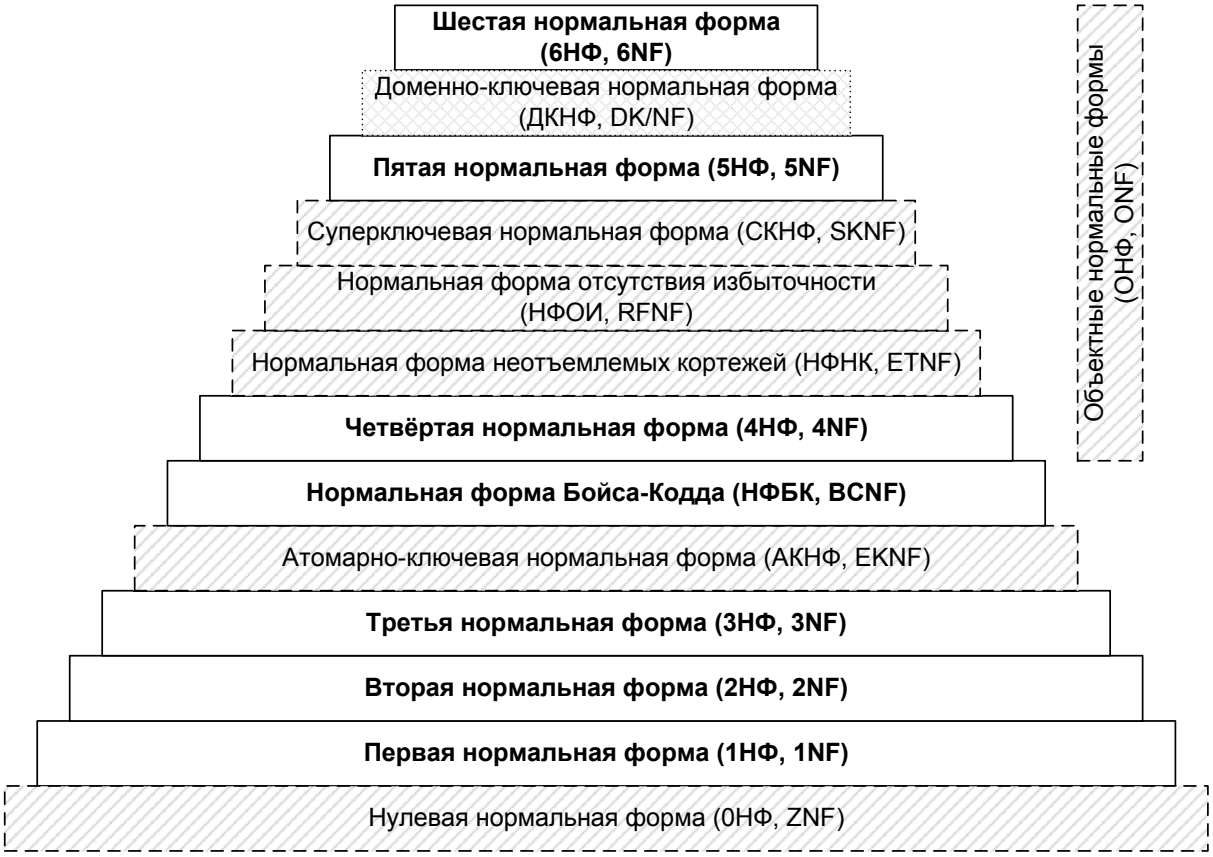


*Рисунок 3.3.t — **Иерархия нормальных форм***

По остальным нормальным формам мы ограничимся лишь весьма краткими определениями (без пояснений) и ссылками на первоисточники.

На вполне закономерный вопрос о том, зачем вообще нужны все эти «неканонические» нормальные формы, есть несколько ответов:

- некоторые из них исторически были сформулированы ранее, чем впоследствии «заменившие» их канонические нормальные формы;
- некоторые из них могут быть рассмотрены как связующие звенья между соседними каноническими нормальными формами;
- некоторые из них показывают те или иные аспекты реляционной теории под другим углом, позволяя лучше и глубже понять рассматриваемые вопросы.

Итак…

Нулевая нормальная форма (0НФ, ZNF) описывает «схему отношения, не находящуюся даже в 1НФ», т.е. в такой схеме допустимы все мыслимые и немыслимые нарушения любых правил и любого здравого смысла.

Атомарно-ключевая нормальная форма (АКНФ, EKNF) (в отличие от 3НФ{252}, см. её каноническую формулировку) требует, чтобы фигурирующий в определении «атрибут A» был не просто ключевым, а атомарно-ключевым, т.е. являлся несократимым детерминантом своей нетривиальной функциональной зависимости.

Нормальная форма неотъемлемых кортежей (НФНК, ETNF) требует, чтобы любое допустимое значение переменной отношения содержало только такие кортежи, удаление хотя бы одного из которых приводило бы к невозможности получить на основе оставшихся кортежей ту же самую информацию, что была до удаления.

Нормальная форма отсутствия избыточности (НФОИ, RFNF) требует, чтобы для любого допустимого значения переменной отношения все возможные проекции этой переменной содержали в себе суперключи исходного отношения, а совокупность этих суперключей составляла заголовок исходного отношения.

Суперключевая нормальная форма (СКНФ, SKNF) требует, чтобы любой компонент любой несократимой зависимости соединения был суперключом исходной переменной отношения.

Особого внимания заслуживает доменно-ключевая нормальная форма, которая, строго говоря, не входит в общую иерархию. Да, логически её удобно размещать между 5НФ и 6НФ (как и сделано на рисунке 3.3.t), но в отличие от всех других нормальных форм, которые «опираются на предыдущие», ДКНФ не «опирается» ни на какие формы (равно как 6НФ не следует из ДКНФ).

Более того, автор ДКНФ показывает[187], что нормальные формы с 1-й по 5-ю сами могут быть следствием ДКНФ.

И совсем отдельного упоминания заслуживают объектные нормальные формы (актуальные для объектно-реляционных баз данных). Существуют 4ОНФ (4-я объектная нормальная форма) и т.д. (потому на рисунке 3.3.t соответствующий блок начинается именно на уровне 4НФ), однако все они выходят за рамки данной книги.

Если вам всё же захотелось подробнее изучить неканонические нормальные формы, то в первую очередь обратитесь к следующим источникам:

- «The New Relational Database Dictionary», C.J. Date (именно отсюда взята общая концепция иерархии нормальных форм и большинство определений);
- «Database Normalization Complete», M. Jason (великолепный краткий справочник по всем мыслимым и немыслимым нормальным формам).

В качестве ещё одного способа запоминания иерархии нормальных форм приведём следующую таблицу[188]:

---

187 «A Normal Form for Relational Databases That Is Based on Domains and Keys», Ronald Fagin.

188 Оригинальная идея взята отсюда: https://en.wikipedia.org/wiki/Database_normalization#Normal_forms

	0НФ	1НФ	2НФ	3НФ	АКНФ	НФБК	4НФ	НФНК	5НФ	ДКНФ	6НФ
Есть первичный ключ	?	+	+	+	+	+	+	+	+	+	+
Нет многозначных атрибутов	?	+	+	+	+	+	+	+	+	+	+
Каждый атрибут атомарен	-	+	+	+	+	+	+	+	+	+	+
Любой неключевой атрибут функционально полно зависит от первичного ключа	-	-	+	+	+	+	+	+	+	+	+
Любой неключевой атрибут нетранзитивно зависит от первичного ключа	-	-	-	+	+	+	+	+	+	+	+
Любой зависимый атрибут входит в состав несократимого ключа	-	-	-	-	+	+	+	+	+	+	+
Любой атрибут нетранзитивно зависит от первичного ключа	-	-	-	-	-	+	+	+	+	+	+
Отсутствуют нетривиальные многозначные зависимости	-	-	-	-	-	-	+	+	+	+	+
Любая зависимость соединения строится на суперключе	-	-	-	-	-	-	-	+	+	+	+
Любая нетривиальная зависимость соединения строится на потенциальном ключе	-	-	-	-	-	-	-	-	+	+	+
Любое ограничение порождено ограничениями доменов и ключей	-	-	-	-	-	-	-	-	-	+	+
Любая зависимость соединения является тривиальной	-	-	-	-	-	-	-	-	-	-	+

Можно сказать, что на текущий момент мы рассмотрели всю теорию, необходимую для выполнения проектирования базы данных, чему и будет посвящён следующий раздел.

Задание 3.3.j: какие из операций нормализации, выполненные вами в заданиях 3.3.a-3.3.i были реально необходимы, а какие носили лишь учебный характер были бы скорее вредны для базы данных, чем полезны? Аргументируйте своё мнение.

ПРОЕКТИРОВАНИЕ БАЗ ДАННЫХ

ПРОЕКТИРОВАНИЕ НА ИНФОЛОГИЧЕСКОМ УРОВНЕ

4.1.1. ЦЕЛИ И ЗАДАЧИ ПРОЕКТИРОВАНИЯ НА ИНФОЛОГИЧЕСКОМ УРОВНЕ

Перед прочтением материала данной главы стоит повторить основы моделирования баз данных{8}.

Как и было сказано в определении инфологического уровня моделирования{10}, основной его целью является создание модели, отражающей сущности предметной области, их атрибуты и связи (возможно, пока не все) между сущностями.

Т.е. необходимо максимально глубоко исследовать предметную область и выразить её понятия в виде сущностей, атрибутов, связей.

Исследование предметной области как раз и представляет основную сложность, разделяющуюся на следующие локальные проблемы:

- извлечение информации;
- глубина исследования;
- границы исследования.

Извлечение информации — универсальная проблема для любого проекта (даже вне контекста баз данных). Как правило[189], у заказчика нет готового полного технического описания проекта, потому необходимо организовать сбор требований к проекту, и в них выделить те требования, которые прямо или косвенно относятся к базе данных.

189 На самом деле — всегда ☺.

Существует множество техник выявления требований (интервью, работа с фокусными группами, анкетирование, семинары и мозговой штурм, наблюдение, прототипирование, анализ документов, моделирование процессов и взаимодействий и т.д. и т.п.[190])

Вне зависимости от выбранного подхода процесс будет долгим, итерационным и направленным на решение главной задачи: собрать максимально корректную информацию в полном объёме. И это приводит нас к следующей локальной проблеме.

Глубина исследования — самая большая сложность для начинающих проектировщиков, что особенно хорошо заметно на примере учебных работ. Недостаточно выявить лишь *некоторые* сущности и *некоторые* их атрибуты. Необходимо выявить их все.

Говоря образно, база данных реальной библиотеки не может состоять из таблиц «книги» (название, год выпуска) и «авторы» (ФИО, год рождения) — в такой базе данных будут десятки таблиц с десятками полей.

Проблема усложняется в силу того, что в общем случае — это не задача заказчика — продумывать все возможные детали, нюансы, аспекты. Это задача именно проектировщика базы данных. И задача весьма серьёзная, т.к. неудача в её решении почти гарантированно приведёт к тому, что база данных будет неадекватна предметной области{12}.

С другой стороны, есть опасность выйти за границы проекта и столкнуться со следующей проблемой.

Границы исследования — проблема часто не техническая, а управленческая, но от этого она не становится проще: в силу настояния заказчика или из технических соображений в базу данных могут начать попадать сущности, вполне реально существующие и связанные с предметной областью, но не актуальные для реализуемого проекта.

Так, продолжая пример с библиотекой, можно от самой базы данных библиотеки перейти к логистике (книги ведь как-то в библиотеку попадают), складскому учёту (книги ведь где-то хранятся), бухгалтерии (ведь финансовые вопросы надо как-то решать), кадровому учёту (ведь в библиотеке работают сотрудники) и т.д.

Чтобы избежать этого эффекта (он называется «размытие границ») применяются специальные техники, не относящиеся к тематике данной книги, однако всегда стоит удерживать в голове, базу данных чего мы проектируем. И поднимать вопрос о том, не вышли ли мы за границы проекта, если возникают сомнения, что те или иные сущности реального мира имеют отношение к проектируемой базе данных.

К перечню задач данного уровня проектирования также можно смело добавить обеспечение соответствия базы данных ключевым требованиям{12}: и пусть их невозможно обеспечить одним лишь проектированием на инфологическом уровне, именно здесь закладывается фундамент для многих решений, которые будут приняты на следующих уровнях и позволят создать действительно эффективную базу данных.

Задание 4.1.а: сформулируйте список вопросов, ответы на которые помогли бы вам улучшить инфологическую схему базы данных «Банк»{408}.

190 См. раздел «2.2.3. Источники и пути выявления требований» в книге «Тестирование программного обеспечения. Базовый курс». (С.С. Куликов) [http://svyatoslav.biz/software_testing_book/]

4.1.2. ИНСТРУМЕНТЫ И ТЕХНИКИ ПРОЕКТИРОВАНИЯ НА ИНФОЛОГИЧЕСКОМ УРОВНЕ

После прочтения предыдущей главы может сложиться впечатление, что для решения столь нетривиальных задач необходимы некие сверхсложные инструменты. Вовсе нет.

Да, непростыми являются сами техники — в первую очередь техники выявления требований{284}, но даже их сложность состоит по большей части в количестве прилагаемых усилий.

Что до инструментов, то они достаточно просты и так или иначе сводятся к записи собранной информации в удобном для восприятия виде. Причём очень важно понимать, что удобно должно быть не только проектировщику базы данных, но и представителю заказчика, с которым результаты проектирования будут многократно обсуждаться.

В плане такого удобства непревзойдёнными остаются всем понятные и привычные многоуровневые списки, которые можно составлять в любом текстовом редакторе. Первым уровнем будут идти сущности, вторым — их атрибуты (см. пример на рисунке 4.1.а).

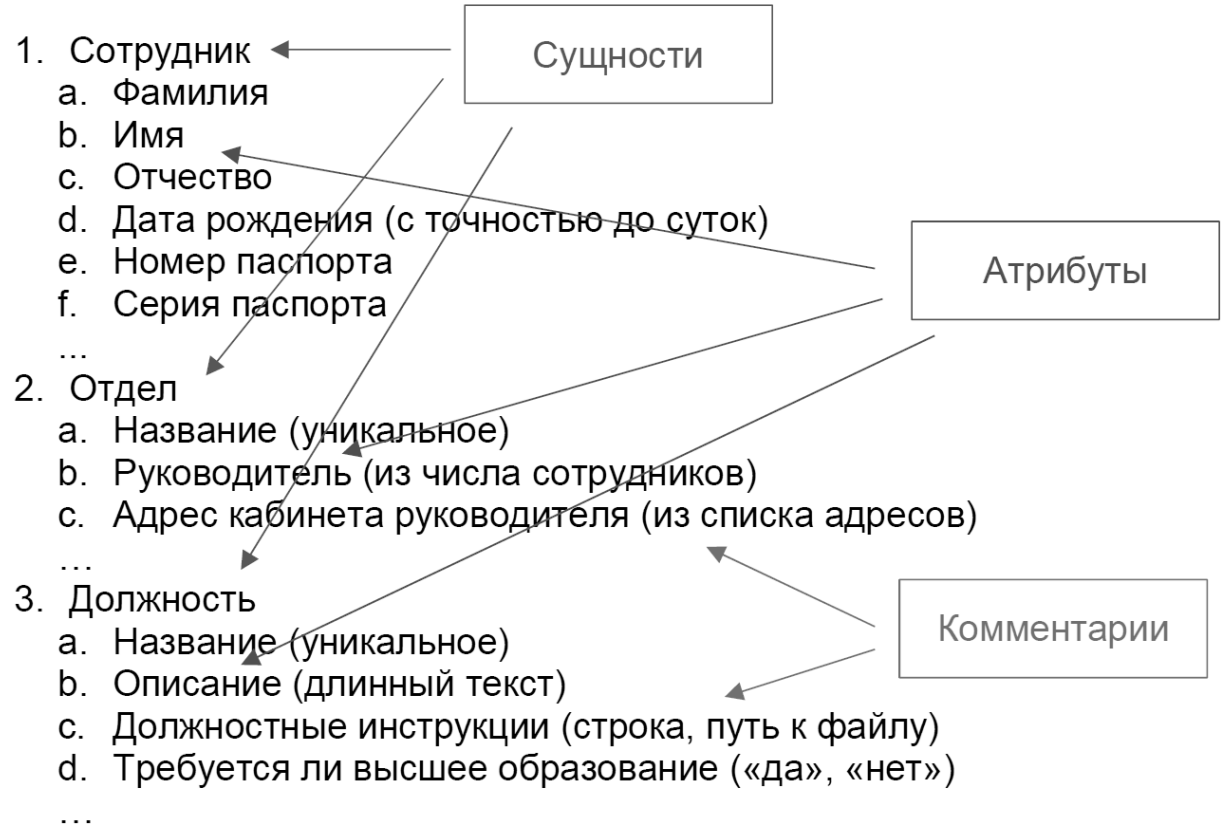


*Рисунок 4.1.а — **Пример представления инфологической модели в текстовом виде***

Текстовое представление удобно не только своей привычностью, но и своей практичностью:

- у всех на компьютере есть необходимое программное обеспечение для просмотра и редактирования текстовых документов;
- создание и правка таких документов происходит очень быстро;
- можно очень удобно располагать пометки и комментарии (которые будут видны сразу без дополнительных действий);
- результат может быть мгновенно распечатан;
- и т.д.

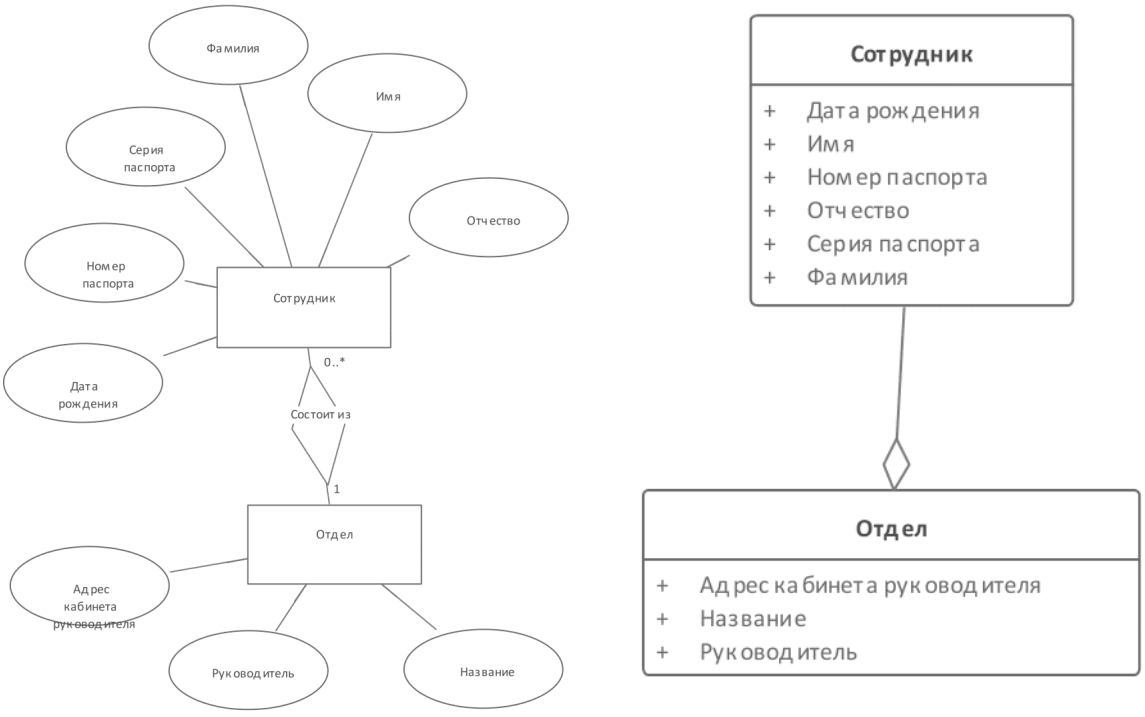
И раз всё так прекрасно, то кажется, что других инструментов не существует просто в силу ненужности. Но это не так.

У текстового представления есть ряд серьёзных недостатков:

- оно неудобно для представления взаимосвязи сущностей (фактически, это можно делать только комментариями);
- оно не компактно (может занимать несколько десятков страниц там, где другие формы представления заняли бы пару экранов);
- оно допускает разночтение технических аспектов реализации базы данных (а чаще всего эти аспекты и вовсе оказываются упущены);
- попытки устранить обозначенные выше недостатки делают текстовое представление перегруженным информацией и постепенно сводят на нет его преимущества.

Альтернативами текстовому представлению являются графические формы — в виде семантических моделей, графовых моделей и UML-диаграмм[191].

Прежде, чем мы рассмотрим их примеры, кратко скажем про упоминаемые в огромном количестве источников ER-диаграммы (entity-relation, сущность-связь). Увы, несмотря на свою теоретическую красоту, они крайне неудобны. Сравните (см. рисунок 4.1.b) представление небольшого фрагмента схемы базы данных в виде ER-диаграммы[192] (в нотации Чена) и в виде UML-диаграммы.



ER-диаграмма UML-диаграмма

*Рисунок 4.1.b — **Сравнение ER-диаграммы и UML-диаграммы***

---

191 Да, существуют специальные языки описания инфологических моделей. Например, ЯИМ (язык инфологического моделирования), но они гармонично сочетают в себе недостатки классического «списочного» подхода и сложность графических моделей. У таких языков есть свои области применения, они имеют право на существование, но назвать их широко распространёнными нельзя, потому в данной книге мы их также не рассматриваем.

192 Даже такое мощнейшее средство проектирования как Sparx Enterprise Architect до сих пор «не умеет» корректно выравнивать надписи внутри элементов ER-диаграмм, что тоже говорит о «востребованности» такой формы представления модели.

Теперь вернёмся к тому, что действительно применяется на практике.

Семантические и графовые модели распространены не так сильно, как UML-диаграммы, но оказываются очень полезными при описании сложных взаимосвязей в предметных областях.

Их сложно использовать «напрямую» для проектирования баз данных (они не проецируются напрямую на реляционную модель), но в качестве постоянно доступной «шпаргалки» они почти незаменимы.

Допустим, нам нужно отразить в базе данных сложную взаимосвязь между ролями сотрудников (для написания контролирующих триггеров{350}). Чтобы не держать такую взаимосвязь в уме, можно выразить её семантической схемой — см. рисунок 4.1.c.

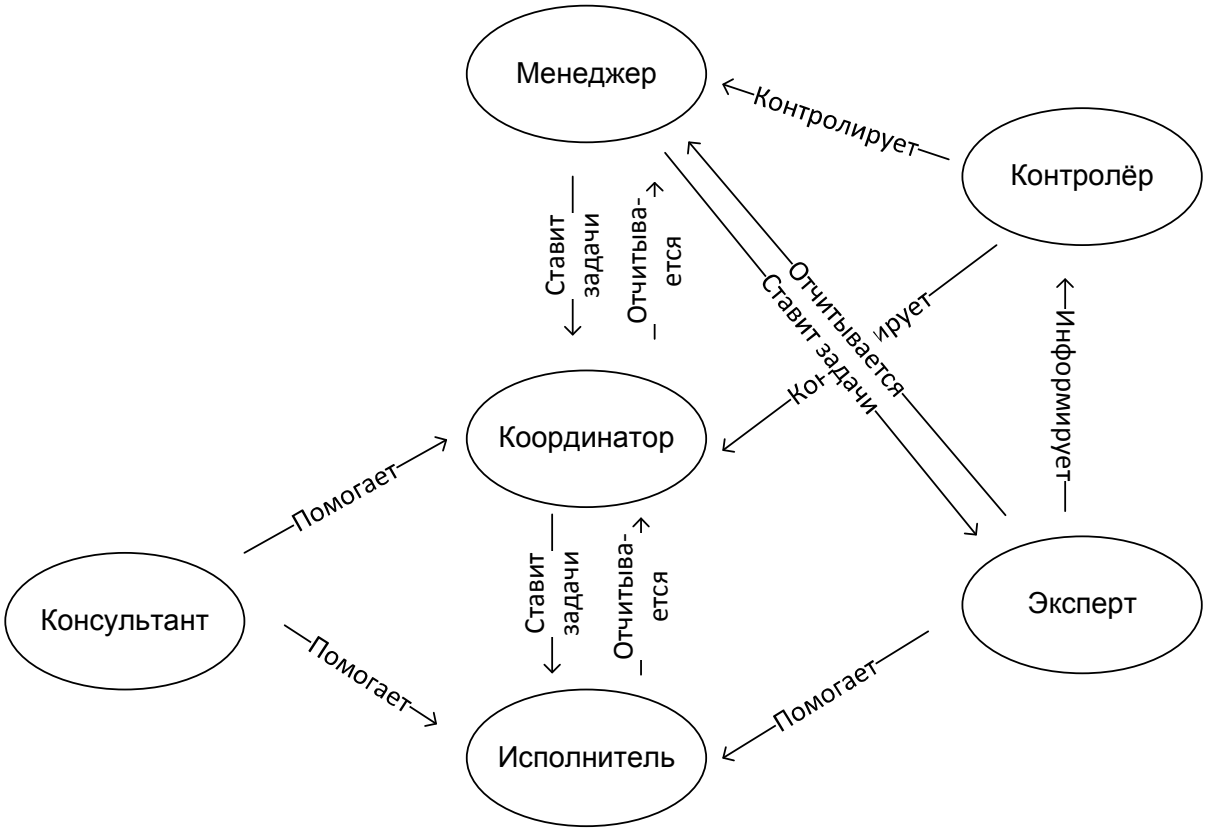


*Рисунок 4.1.c — **Пример семантической схемы***

Графовые модели очень удобны для описания состояний и переходов между ними. Опять же, намного проще нарисовать схему (см. рисунок 4.1.d), чем постоянно держать это в уме.

Семантические и графовые модели могут быть полезны не только для описания сложных и необычных предметных областей, но и таких, в которых всё кажется привычным и очевидным, однако в данном конкретном проекте имеет некие нетипичные особенности (например, на рисунке 4.1.c показано, что менеджеру консультант не помогает, хотя интуитивно кажется, что должен помогать; на рисунке 4.1.d некая задача не может переходить между состояниями «В работе» и «Отклонено», хотя интуитивно и тут не видится никаких объективных запретов).

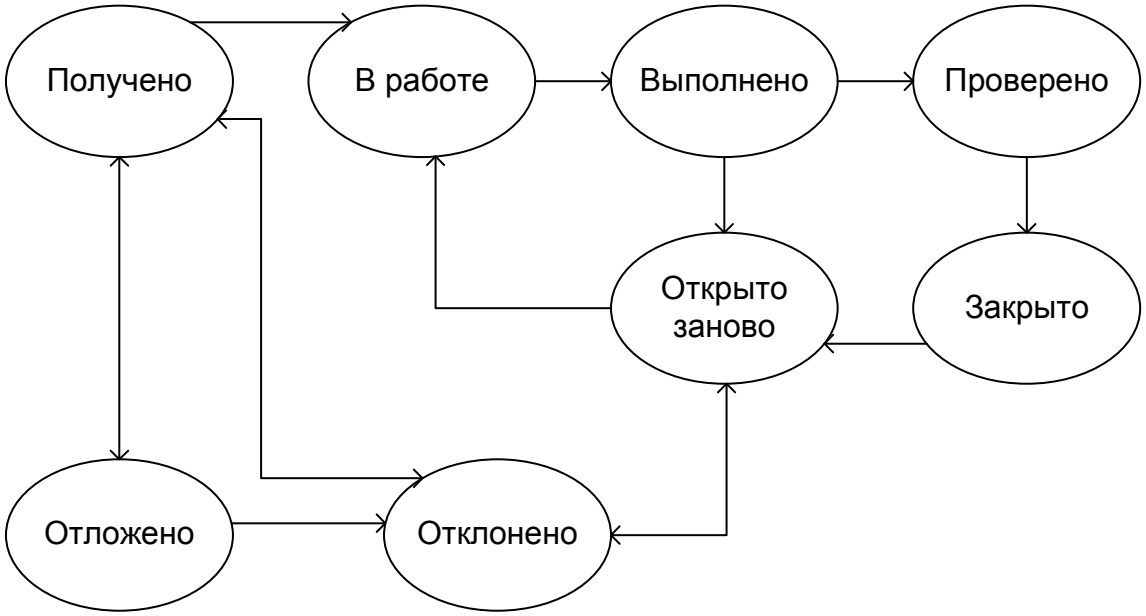


*Рисунок 4.1.d — **Пример графовой схемы***

Наконец переходим к UML и начнём с краткого напоминания видов связей[193] (их общий перечень показан на рисунке 4.1.e). Да, возможности UML значительно шире, но для моделирования баз данных в общем случае достаточно будет помнить, что такое «связь», «класс», «атрибут», «метод».

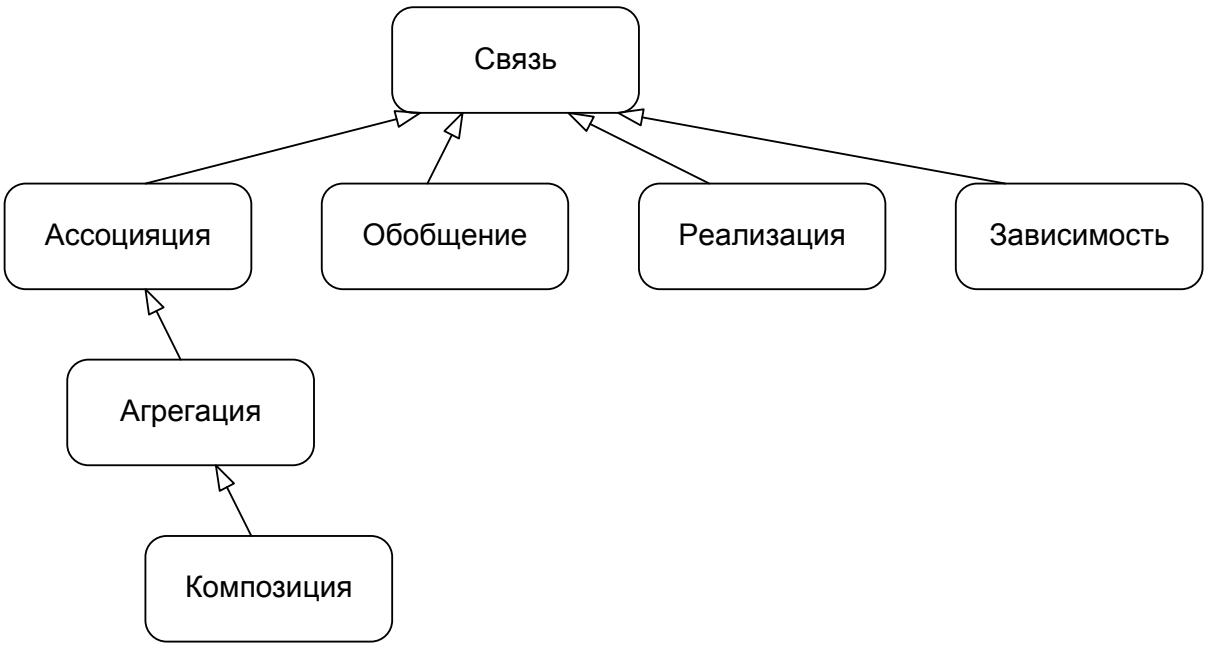


*Рисунок 4.1.e — **Иерархия UML-связей***

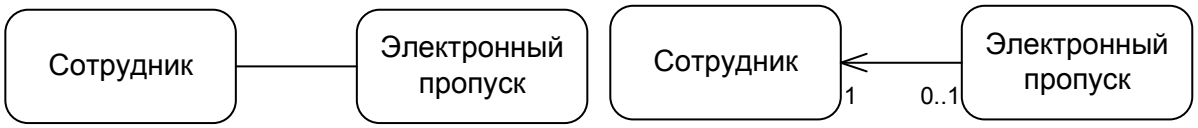
[193] Полноценное рассмотрение данной технологии выходит за рамки этой книги, но в Интернет есть большое количество хорошей документации. Начать можно отсюда: https://www.tutorialspoint.com/uml/

Рассмотрим подробнее все виды UML-связей и их применение в моделировании баз данных.

Ассоциация — самый общий, универсальный и «ни к чему не обязывающий вариант»: просто отражается тот факт, что некие сущности находятся во взаимосвязи (при этом вид и особенности этой взаимосвязи не конкретизируются, хоть при желании и можно указать некоторые параметры — например, мощность связи и её направление).

В примере ниже (рисунок 4.1.f) показано, что сотрудник и электронный пропуск связаны между собой. В примере с уточнёнными параметрами показано, что:

- электронный пропуск принадлежит сотруднику (а не наоборот);
- электронный пропуск обязан принадлежать ровно одному сотруднику;
- у сотрудника может не быть электронного пропуска;
- у сотрудника может быть не более одного электронного пропуска.



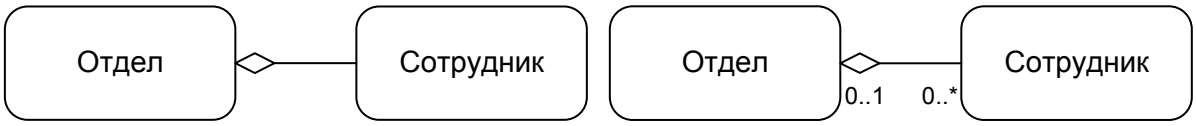
Самый общий вид Уточнение некоторых параметров

*Рисунок 4.1.f — **Пример ассоциации***

Агрегация — частный случай ассоциации, показывающий, что дочерние элементы входят в состав родительского элемента (но могут и существовать отдельно).

В примере ниже (рисунок 4.1.g) показано, что отдел состоит из сотрудников («агрегирует» собой сотрудников). В примере с уточнёнными параметрами показано, что:

- в отделе может быть от нуля до бесконечности сотрудников;
- сотрудник может не принадлежать никакому отделу;
- сотрудник может принадлежать не более, чем одному отделу.



Самый общий вид Уточнение некоторых параметров

*Рисунок 4.1.g — **Пример агрегации***

Композиция — ещё один частный случай ассоциации, показывающий, что дочерние элементы входят в состав родительского элемента, но, в отличие от агрегации, здесь дочерние элементы не могут существовать самостоятельно (без родительского элемента).

В примере ниже (рисунок 4.1.h) показано, что текст входит в состав музыкального произведения (и сам по себе не существует, т.е. это именно «текст песни»). В примере с уточнёнными параметрами показано, что:

- каждый текст обязан относиться хотя бы к одному музыкальному произведению;
- текст может относиться к более, чем одному музыкальному произведению;
- к каждому музыкальному произведению может относиться от нуля до бесконечности текстов.

Самый общий вид Уточнение некоторых параметров

*Рисунок 4.1.h — **Пример композиции***

Обобщение — вид связи, показывающий, что её дочерний элемент является частным случаем родительского.

Обобщение очень часто используется в программировании (класс родитель и класс-потомок как частный случай класса-родителя), а в проектировании баз данных обычно сразу выражается в виде той или иной ассоциации, но на начальных этапах проектирования инфологического уровня обобщения имеют право на существование, т.к. могут отражать реальное положение дел в предметной области.

В примере ниже (рисунок 4.1.i) показано, что контрактор является частным случаем сотрудника. Также именно обобщение использовано в рисунке 4.1.e для иллюстрации иерархии UML-связей. Обратите внимание, что у обобщения не бывает мощности, т.к. выражения вида «сотрудник — это N контракторов» лишено практического смысла, здесь показана именно иерархия, которая по определению не имеет такого свойства как «мощность связи».

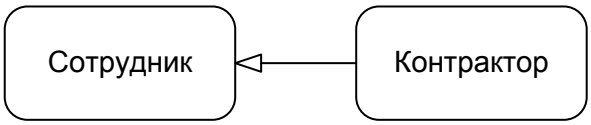


*Рисунок 4.1.i — **Пример обобщения***

Реализация — вид связи, показывающий, что дочерний элемент реализует поведение, задаваемое родительским элементом.

Как и обобщение, реализация широко используется в программировании (интерфейс и реализующие его классы), а в проектировании баз данных она выражается в виде той или иной ассоциации, но, опять же, на начальных этапах проектирования инфологического уровня реализация может быть применена в UML-диаграмме для отражения особенностей предметной области.

В примере ниже (рисунок 4.1.j) показано, что техническое решение подчинено стандарту, т.к. обязано «вести себя» так, как сказано в стандарте.

Несмотря на то, что многие инструменты позволяют выставить «мощность связи реализации», в классическом варианте на UML-схемах она не указывается.

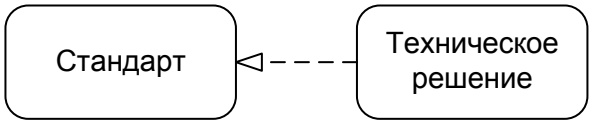
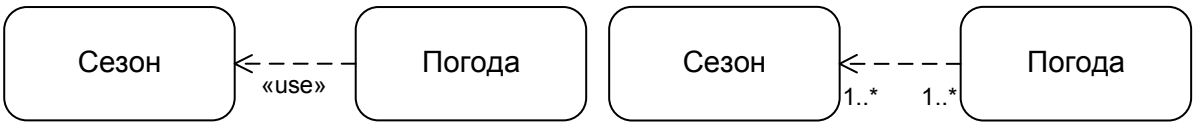


*Рисунок 4.1.j — **Пример реализации***

Зависимость — вид связи, показывающий, что изменения в родительском элементе обязательно приводят к изменениям в дочернем элементе (но не наоборот).

В примере ниже (рисунок 4.1.k) показано, что изменение сезона должно приводить к изменению погоды (обратное — неверно). В примере с уточнёнными параметрами показано, что:

- у каждого сезона есть хотя бы один вид погоды (но может быть и больше, до бесконечности);
- каждый вид погоды обязан относиться хотя бы к одному сезону (но может относиться и к большему количеству сезонов, до бесконечности).



Самый общий вид Уточнение некоторых параметров

*Рисунок 4.1.k — **Пример зависимости***

Осталось рассмотреть, как UML-понятия «класс», «атрибут» и «метод» связаны с проектированием баз данных.

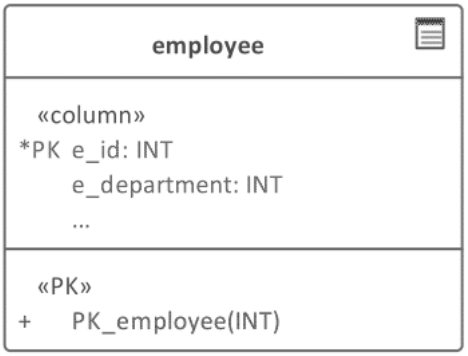


*Рисунок 4.1.l — **Связь UML-понятий «класс», «атрибут» и «метод» с проектированием баз данных***

Здесь для иллюстрации будет достаточного одного рисунка 4.1.l — прямоугольник обозначает класс, в верхнем блоке приведено имя класса, в среднем — атрибуты класса, а в нижнем — методы класса.

Проще всего разобраться с самим «классом» (в проектировании баз данных это будет «схема отношения» или просто «отношение» в зависимости от контекста) и «атрибутом» (в проектировании баз данных это будет «атрибут отношения»).

С методами ситуация выглядит чуть необычнее, но она идеально отражает логику работы СУБД: к «методам отношения» будут относиться первичный ключ, индексы и иные *операции*.

И это не опечатка — это именно операции. С точки зрения человека, например, первичный ключ — это просто поле или несколько полей, обладающих определёнными свойствами, но с точки зрения СУБД — это ещё и необходимость *выполнять* проверку уникальности значений. Аналогично, индекс — это необходимость *выполнять* набор действий по его построению и актуализации. Потому это — именно *операции*, и они отражаются в блоке методов (т.е. того, что отношение «умеет делать»).

Можно смело утверждать, что такого краткого рассмотрения UML будет достаточно для понимания всех примеров, представленных в этой книге.

Существует очень много инструментов, используемых для проектирования моделей баз данных в UML-нотации, они интенсивно развиваются, и описание работы с ними почти всегда можно найти в прилагаемом руководстве пользователя.

Отметим лишь, что одним из наиболее популярных является Sparx Enterprise Architect, в котором и созданы все UML-диаграммы, представленные в данной книге.

Переходим к рассмотрению большого примера.

Задание 4.1.b: на основе инфологической схемы базы данных «Банк»{408} создайте соответствующую семантическую схему.

4.1.3. ПРИМЕР ПРОЕКТИРОВАНИЯ НА ИНФОЛОГИЧЕСКОМ УРОВНЕ

До сих пор для иллюстрации тех или иных идей были использованы простые не связанные между собой примеры, в которых чаще всего были задействованы одна-две схемы отношения.

Но с этого момента мы приступаем к рассмотрению «долгоиграющего» и очень объёмного примера: мы будем проектировать полноценную базу данных — пусть и учебную, но максимально приближенную к реальности.

Итак, представим, что нам предстоит создать файлообменный сервис, т.е. веб-приложение, в котором пользователи могут размещать файлы для скачивания другими пользователями.

Осознанно до предела упростим[194] перечень и форму представления требований к приложению, получив в первом приближении такой результат:

1. Приложение содержит несколько страниц (их количество, иерархия, информационное наполнение может быть произвольным).
2. У приложения есть пользователи, которые могут создавать группы и объединяться в такие группы.
3. У каждого пользователя может быть несколько ролей, наделённых набором прав.
4. Пользователи могут закачивать и скачивать файлы, открывая к ним доступ для отдельных пользователей, для групп пользователей и «внешний» доступ.
5. Пользователи могут оставлять комментарии к представленным на сервера файлам.
6. У файла должен быть рейтинг.
7. Можно «отвечать на комментарии» (до глубины вложенности в 10 уровней).
8. Каждый файл обязательно должен относиться к одной из категорий файлов, определяющих перечень распространяемых на файл ограничений.
9. Приложение должно вести протокол всех действий всех пользователей.
10. Должна быть предусмотрена возможность блокировки пользователей, групп пользователей и незарегистрированных посетителей (по ip).
11. Приложение должно с минимальными временными задержками показывать текущую статистику по количеству пользователей, количеству и объёму файлов, количеству и объёму скачанных файлов.

Рассмотрим перечень сущностей, необходимость в которых проистекает из наличия таких требований:

№ тр.	Текст требования	Список сущностей
1	Приложение содержит несколько страниц (их количество, иерархия, информационное наполнение может быть произвольным)	Страница.
2	У приложения есть пользователи, которые могут создавать группы и объединяться в такие группы.	Пользователь. Группа пользователей.
3	У каждого пользователя может быть несколько ролей, наделённых набором прав.	Роль. Право.
4	Пользователи могут закачивать и скачивать файлы, открывая к ним доступ для отдельных пользователей, для групп пользователей и «внешний» доступ.	Файл.

[194] См. раздел «2.2. Тестирование документации и требований» в книге «Тестирование программного обеспечения. Базовый курс». (С.С. Куликов) [http://svyatoslav.biz/software_testing_book/]

5	Пользователи могут оставлять комментарии к представленным на сервера файлам.	Комментарий.
6	У файла должен быть рейтинг.	Оценка.
7	Можно «отвечать на комментарии» (до глубины вложенности в 10 уровней).	Комментарий.
8	Каждый файл обязательно должен относиться к одной из категорий файлов, определяющих перечень распространяемых на файл ограничений.	Категория файлов.
9	Приложение должно вести протокол всех действий всех пользователей.	Протокол.
10	Должна быть предусмотрена возможность блокировки пользователей, групп пользователей и незарегистрированных посетителей (по ip).	Причина блокировки.
11	Приложение должно с минимальными временными задержками показывать текущую статистику по количеству пользователей, количеству и объёму файлов, количеству и объёму скачанных файлов.	Статистика.

Итак, мы получили следующий перечень сущностей:

1. Страница.
2. Пользователь.
3. Группа пользователей.
4. Роль.
5. Право.
6. Файл.
7. Оценка.
8. Комментарий.
9. Категория файлов.
10. Протокол.
11. Причина блокировки.
12. Статистика.

Здесь пока не учтены «технические отношения» (для связей «многие ко многим»). Также здесь вполне могут появиться и иные сущности, которые пока не были учтены.

К сожалению, в формате книги невозможно передать процесс общения с заказчиком (фактически, это, чаще всего, обычная беседа в формате «вопрос-ответ»), но предположим, что после общения и прояснения подробностей мы получили следующую картину (добавились атрибуты сущностей, самих сущностей стало больше):

1. Страница:
   a. Родительская страница.
   b. Название страницы (для отображения на самой странице).
   c. Имя страницы в меню.
   d. Заглавие страницы (HTML-тег TITLE).
   e. Ключевые слова (HTML-тег meta … keywords).
   f. Описание страницы (HTML-тег meta … description).
   g. Текстовое наполнение страницы.
2. Пользователь:
   a. Логин.
   b. Пароль.
   c. E-mail.
   d. Дата и время регистрации.

   e. Дата рождения.
   f. Бонус по количеству закачанного.
3. Группа пользователей:
   a. Владелец (создатель) группы.
   b. Название группы.
   c. Описание группы.
4. Роль:
   a. Название роли.
   b. Ограничения по объёму закачиваемых файлов.
   c. Ограничения по объёму скачиваемых файлов.
   d. Ограничения по количеству закачиваемых файлов.
   e. Ограничения по количеству скачиваемых файлов.
   f. Ограничения по скорости скачивания.
5. Право:
   a. Название права.
   b. Описание права.
6. Файл:
   a. Размер (в байтах).
   b. Дата добавления.
   c. Срок хранения (до какой даты и какого времени).
   d. Исходное имя (без расширения).
   e. Исходное расширение.
   f. Имя на сервере.
   g. Контрольная сумма.
   h. Ссылка на удаление (для незарегистрированных пользователей).
7. Ссылка на «публичное» скачивание файла:
   a. К какому файлу.
   b. Значение ссылки.
   c. Срок действия ссылки (дата и время, после которых ссылка считается недействительной и должна быть удалена).
   d. Пароль на скачивание (если есть).
8. Параметр доступа к файлу:
   a. К какому файлу.
   b. Какое действие.
   c. Кому разрешено.
9. Оценка:
   a. Какой файл оценивается.
   b. Кто оценивает.
   c. Значение оценки.
10. Комментарий:
   a. К какому файлу.
   b. К какому комментарию.
   c. Текст комментария.
   d. Кто оставил комментарий.
   e. Дата и время добавления комментария.
11. Категория файлов:
   a. Название категории.
   b. Возрастные ограничения (если есть).
12. Возрастное ограничение:
   a. Минимальный возраст (в годах).
   b. Название ограничения.
   c. Описание ограничения.

13. Протокол:
    a. Пользователь.
    b. Ip-адрес.
    c. Операция.
    d. Файл (если задействован).
    e. Дата и время выполнения действия.
    f. Параметры действия.
14. Архив протокола (для записей старше месяца):
    a. Пользователь.
    b. Ip-адрес.
    c. Операция.
    d. Файл (если задействован).
    e. Дата и время выполнения действия.
    f. Параметры действия.
15. Причина блокировки:
    a. Название причины.
    b. Описание причины.
16. Чёрный список IP-адресов:
    a. IP-адрес в формате IPv4.
    b. IP-адрес в формате IPv6.
    c. Дата и время, до которых действительна блокировка.
    d. Причина блокировки.
17. Статистика:
    a. Всего зарегистрированных пользователей.
    b. Пользователей зарегистрировалось сегодня.
    c. Всего закачано файлов.
    d. Файлов закачано сегодня.
    e. Общий объём закачанных файлов.
    f. Объём файлов, закачанных сегодня.
    g. Всего скачано файлов.
    h. Файлов скачано сегодня.
    i. Общий объём скачанных файлов.
    j. Объём файлов, скачанных сегодня.

И снова предположим, что, обдумав и обсудив с коллегами полученный результат, мы через пару дней вернулись к обсуждению с заказчиком, в процессе которого постепенно начали прописывать технические детали.

Получилось следующее:

1. Страница:
    a. Идентификатор.
    b. Родительская страница (rFK).
    c. Название страницы (для отображения на самой странице).
    d. Имя страницы в меню (уникальное в рамках одного уровня меню).
    e. Заглавие страницы (HTML-тег TITLE).
    f. Ключевые слова (HTML-тег meta … keywords).
    g. Описание страницы (HTML-тег meta … description).
    h. Текстовое наполнение страницы.
2. Пользователь:
    a. Идентификатор.
    b. Логин (уникальный).
    c. Пароль (sha256-хэш).

   d. E-mail (уникальный).
   e. Дата и время регистрации (с точностью до секунды).
   f. Дата рождения (с точностью до дня).
   g. Бонус в виде скорости по количеству закачанного (FK).
   h. Срок действия бонуса в виде скорости по количеству закачанного (до какой даты и времени, с точностью до секунды).
   i. Бонус в виде объёма по количеству закачанного (FK).
   j. Срок действия бонуса в виде объёма по количеству закачанного (до какой даты и времени, с точностью до секунды).
3. Бонус:
   a. Идентификатор.
   b. За какое количество закачанных файлов выдаётся.
   c. За какой объём закачанных файлов выдаётся.
   d. Сколько добавляет к скорости скачивания.
   e. Сколько добавляет к объёму скачивания.
4. Группа пользователей:
   a. Идентификатор.
   b. Владелец (создатель) группы (FK).
   c. Название группы (уникальное).
   d. Описание группы.
5. Роль:
   a. Идентификатор.
   b. Название роли (уникальное).
   c. Ограничения по объёму закачиваемых файлов (в байтах).
   d. Ограничения по объёму скачиваемых файлов (в байтах).
   e. Ограничения по количеству закачиваемых файлов.
   f. Ограничения по количеству скачиваемых файлов.
   g. Ограничения по скорости скачивания (в байтах в секунду).
6. Право:
   a. Идентификатор.
   b. Название права (уникальное).
   c. Описание права.
7. Файл:
   a. Идентификатор.
   b. Размер (в байтах).
   c. Дата и время добавления (с точностью до секунды).
   d. Срок хранения (до какой даты и какого времени, с точностью до секунды).
   e. Исходное имя (без расширения).
   f. Исходное расширение.
   g. Имя на сервере (sha256-хэш).
   h. Контрольная сумма (sha256-хэш).
   i. Ссылка на удаление (для незарегистрированных пользователей, sha256-хэш).
8. Ссылка на «публичное» скачивание файла:
   a. Идентификатор.
   b. К какому файлу (FK).
   c. Значение ссылки (sha256-хэш).
   d. Срок действия ссылки (дата и время, после которых ссылка считается недействительной и должна быть удалена, с точностью до секунды).
   e. Пароль на скачивание (если есть, sha256-хэш).

9. Параметр доступа к файлу:
   a. Идентификатор.
   b. К какому файлу (FK).
   c. Какое действие (FK).
   d. Какому пользователю разрешено (FK).
   e. Какой группе разрешено (FK).
   f. Признак «разрешено всем зарегистрированным пользователям».
   g. Признак «разрешено вообще всем».
10. Оценка:
   a. Идентификатор.
   b. Какой файл оценивается (FK).
   c. Какой пользователь оценивает (FK).
   d. Значение оценки (от 1 до 10).
11. Комментарий:
   a. Идентификатор.
   b. К какому файлу (FK).
   c. К какому комментарию (FK).
   d. Текст комментария.
   e. Какой пользователь оставил комментарий (FK).
   f. Дата и время добавления комментария (с точностью до секунды).
12. Категория файлов:
   a. Идентификатор.
   b. Название категории (уникальное).
   c. Возрастные ограничения (если есть, FK).
13. Возрастное ограничение:
   a. Идентификатор.
   b. Минимальный возраст (в годах).
   c. Название ограничения (уникальное).
   d. Описание ограничения.
14. Протокол:
   a. Пользователь (FK, NULL для незарегистрированных).
   b. Ip-адрес.
   c. Операция (FK).
   d. Файл (если задействован, FK).
   e. Дата и время выполнения действия (с точностью до секунды).
   f. Параметры действия (если есть, текст).
15. Операция:
   a. Идентификатор.
   b. Название (уникальное).
16. Архив протокола (для записей старше месяца):
   a. Пользователь (FK, NULL для незарегистрированных).
   b. Ip-адрес.
   c. Операция (FK).
   d. Файл (если задействован, FK).
   e. Дата и время выполнения действия (с точностью до секунды).
   f. Параметры действия (если есть, текст).
17. Причина блокировки:
   a. Идентификатор.
   b. Название причины (уникальное).
   c. Описание причины.

18. Чёрный список IP-адресов:
    a. IP-адрес в формате IPv4 (значение уникально).
    b. IP-адрес в формате IPv6 (значение уникально).
    c. Дата и время, до которых действительна блокировка (с точностью до секунды).
    d. Причина блокировки (FK).
19. Статистика:
    a. Всего зарегистрированных пользователей.
    b. Пользователей зарегистрировалось сегодня.
    c. Всего закачано файлов.
    d. Файлов закачано сегодня.
    e. Общий объём закачанных файлов (в байтах).
    f. Объём файлов, закачанных сегодня (в байтах).
    g. Всего скачано файлов.
    h. Файлов скачано сегодня.
    i. Общий объём скачанных файлов (в байтах).
    j. Объём файлов, скачанных сегодня (в байтах).

Как вы можете сами легко убедиться, текстовое описание уже занимает почти три страницы, и здесь ведь ещё не отражены связи и промежуточные отношения для связей «многие ко многим». Мы пока не отказываемся от текстового представления, но дополним его графическим, чтобы продемонстрировать, насколько оно компактнее и нагляднее (см. рисунок 4.1.m).

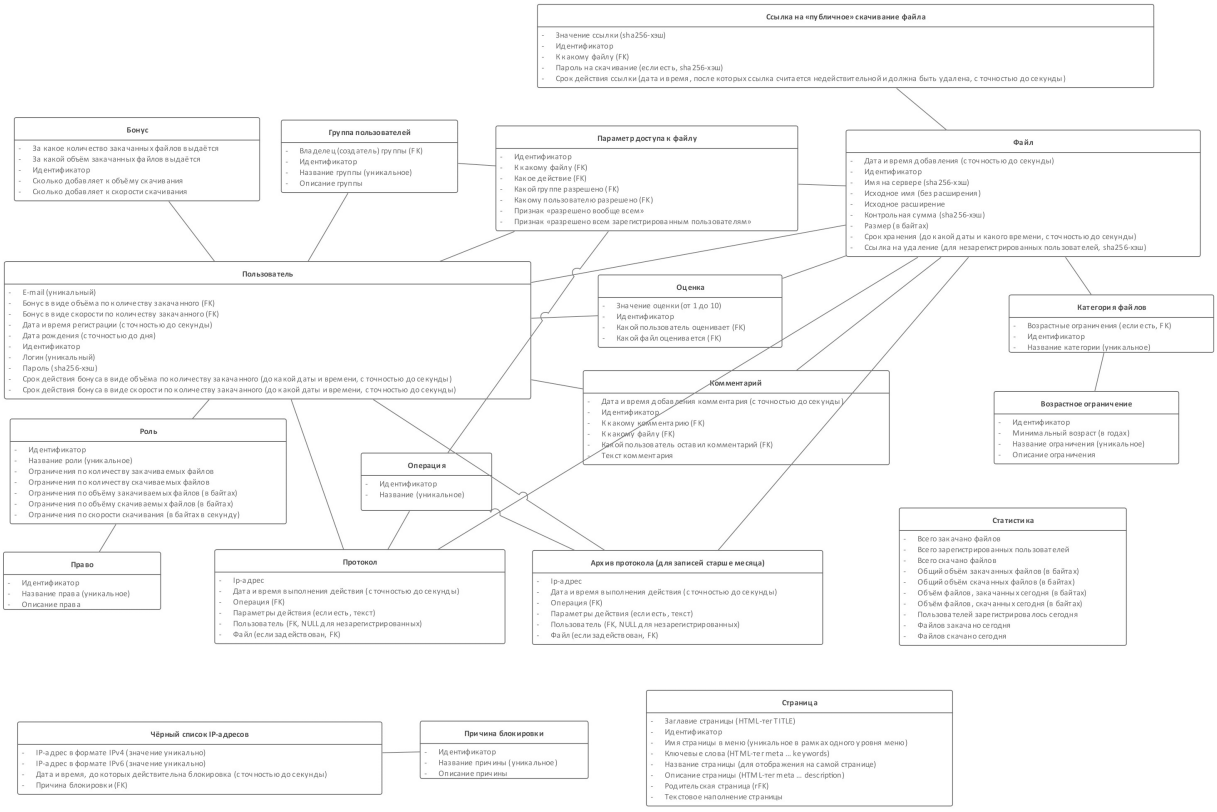


*Рисунок 4.1.m — **Графическое представление результатов проектирования на инфологическом уровне***[195]

195 Полноразмерную картинку вы можете найти в раздаточном материале{5} или по этой ссылке: http://svyatoslav.biz/relational_databases_book_download/pics/conceptual_level_graphical_representaion.png.

Вместо трёх страниц текста мы получили одну картинку (которая помещается на один экран), на которой не только отображена вся та же самая информация, но ещё и добавлены связи между сущностями, что даёт нам дополнительную информацию и повышает наглядность.

В данном конкретном примере была применена небольшая «хитрость»: пусть мы и находимся сейчас на инфологическом уровне, но и в текстовое описание, и в графическое представление мы уже добавили искусственные первичные ключи{42} (поля «Идентификатор») и внешние ключи{47} — что, говоря строго формально, относится к даталогическому уровню, но уже сейчас очень помогает с технической точки зрения.

Эта «хитрость» наглядно иллюстрирует, что разбиение на уровни моделирования во многом носит условный характер, а сами границы между уровнями являются очень размытыми{9}.

Также эта «хитрость» даёт нам ещё одно преимущество: если внимательно вчитаться в формулировки описаний атрибутов сущностей, можно заметить, что это — почти готовые комментарии к полям таблиц, которые мы будем проектировать уже очень скоро.

Чем сложнее проект, сложнее его база данных и сложнее предметная область, тем больше времени необходимо провести в общении с заказчиком для того, чтобы максимально полно выявить все сущности, атрибуты, связи и особенности бизнес-процессов, которые окажут влияние на схему базы данных (если этого не сделать, очень высока вероятность нарушить такое ключевое свойство базы данных как адекватность предметной области{12}).

Однако всё же стоит помнить, что на этапе проектирования на инфологическом уровне всё равно не получится учесть все нюансы (и именно поэтому нам придётся несколько раз пройти весь процесс проектирования «сверху-вниз{11}» и «снизу вверх{12}»).

Потому сейчас мы можем смело переходить к следующему этапу — проектированию на даталогическом уровне.

Задание 4.1.c: доработайте инфологическую схему базы данных «Банк»{408} так, чтобы устранить все обнаруженные в ней недостатки (насколько это возможно без привлечения «гипотетического заказчика» для получения ответов на возникающие вопросы).

4.2. ПРОЕКТИРОВАНИЕ НА ДАТАЛОГИЧЕСКОМ УРОВНЕ

4.2.1. ЦЕЛИ И ЗАДАЧИ ПРОЕКТИРОВАНИЯ НА ДАТАЛОГИЧЕСКОМ УРОВНЕ

Перед прочтением материала данной главы стоит повторить основы моделирования баз данных{8}.

Как и было сказано в определении даталогического уровня моделирования{10}, основной его целью является детализация инфологической модели и превращение её в схему, на которой ранее выявленные сущности, атрибуты и связи оформляются согласно правилам моделирования для выбранного вида базы данных (часто даже с учётом конкретной СУБД).

Здесь не прекращается анализ предметной области (скорее всего, появятся новые вопросы, ответы на которые вынудят нас внести изменения в инфологическую модель), но особое внимание на данном уровне стоит уделить уже рассмотренным ранее требованиям, предъявляемым к любой базе данных{12}.

Поскольку именно на даталогическом уровне формируется будущая структура базы данных, все принятые здесь решения (и, увы, допущенные ошибки) будут очень сильно влиять на степень адекватности базы данных предметной области, на удобство использования базы данных, на её производительность и защищённость её данных.

Фактически, вся информация, представленная ранее в разделах 2{21} и 3{161} данной книги, в полной мере относится к особенностям проектирования на даталогическом уровне.

Какие таблицы{21} создать? Почему именно такие? Может быть, есть лучший вариант? Какие у этих таблиц будут поля{21} (а также какие у этих полей будут типы данных и иные свойства)? Какие ключи{35} использовать? Какие связи{57} и как именно устанавливать? Может быть, уже видна часть индексов{106} и представлений{340}? Достаточно ли наша схема базы данных нормализована{211}? Не подвержена ли она каким-то аномалиям{161}? На все эти и многие другие вопросы придётся искать ответы в процессе проектирования базы данных на даталогическом уровне.

Если свести вышесказанное буквально к двум фразам, то получится:

- цель проектирования на даталогическом уровне — создать структуру будущей базы данных;
- задача проектирования на даталогическом уровне — сделать создаваемую структуру максимально качественной, избавленной от обнаружимых на данном этапе проблем.

Предвидеть и заранее устранить все проблемы всё равно не получится — и это совершенно нормально. Но в наших силах — хотя бы избавиться от наиболее очевидных, типичных ошибок.

Достичь этого во многом помогает т.н. «восходящее проектирование{12}»: мы уже видим структуру базы данных и уже можем анализировать, к каким результатам приведёт выполнение тех или иных запросов на такой структуре. И пока «дела не зашли слишком далеко», мы можем легко изменить структуру базы данных, если видим, что она обладает тем или иным недостатком.

И кроме представленной в разделах 2{21} и 3{161} данной книги общей информации, мы можем применять различные специфические техники и инструменты, которые могут сильно упростить нашу работу.

Рассмотрим их.

Задание 4.2.а: сформулируйте список вопросов, ответы на которые помогли бы вам улучшить даталогическую схему базы данных «Банк»{408}.

4.2.2. ИНСТРУМЕНТЫ И ТЕХНИКИ ПРОЕКТИРОВАНИЯ НА ДАТАЛОГИЧЕСКОМ УРОВНЕ

Прежде, чем приступить к рассмотрению непосредственно инструментов и техник, сделаем небольшое отступление и подчеркнём, что необходимо знать и уметь для того, чтобы успешно проектировать базы данных на даталогическом уровне.

В идеале	Как минимум
Глубоко понимать реляционную теорию.	Иметь представление о реляционной теории.
Знать «на уровне инстинктов» теорию нормализации.	Понимать хотя бы 1-3 нормальные формы.
Глубоко знать SQL.	Знать основы SQL.
Знать целевую СУБД на уровне, позволяющем проводить её тонкое администрирование.	Уметь устанавливать и настраивать целевую СУБД.
Уметь использовать средства проектирования.	Иметь много терпения для изучения средств проектирования.

Реляционная теория и теория нормализации в достаточной мере представлены в предыдущих разделах данной книги, для глубокого погружения в SQL вы можете обратиться к книге «Работа с MySQL, MS SQL Server и Oracle в примерах»[196].

Сложнее всего будет с последними двумя пунктами — и СУБД, и средства проектирования развиваются слишком стремительно, чтобы по ним существовали некие «фундаментальные» книги (они будут устаревать уже к моменту публикации), потому самое надёжное решение — читать официальную документацию по конкретной версии конкретной СУБД и конкретного средства проектирования, которые вы используете. Возможно, официальная документация и не будет написана предельно упрощённым языком, но она будет как минимум актуальной — и это преимущество в данном случае перевешивает все недостатки.

Вернёмся к теме данной главы.

Проектирование на инфологическом уровне предполагало интенсивное обсуждение формируемой модели с заказчиком, и потому там мы старались применять такие средства, которые были бы доступны и понятны «не техническому» человеку (который, например, может быть глубочайшим экспертом в международной логистике или иной предметной области, но совершенно не обязан понимать технические тонкости работы баз данных).

Начиная с даталогического уровня, мы уже в куда большей степени ориентируемся на технических специалистов, поэтому используемые здесь решения могут быть непонятны заказчику — и это нормально, т.к. мы будем вносить необходимые правки в инфологическую модель и продолжать говорить с заказчиком «на его языке», не перегружая его техническими подробностями.

И первое, что стоит сделать техническим специалистам — это прийти к нескольким ключевым соглашениям, которые в общем случае можно поделить на две группы:

- Соглашения относительно СУБД.
- Соглашения относительно БД.

Отсутствие или недостаточная проработанность подобных соглашений очень сильно снижает такое свойство базы данных как удобство использования{14}.

196 «Работа с MySQL, MS SQL Server и Oracle в примерах» (С.С. Куликов) [http://svyatoslav.biz/database_book/]

Соглашения относительно СУБД могут включать в себя, например, следующие пункты (конкретные примеры будут в следующем разделе{307}):

- Вид СУБД.
- Конкретный продукт (конкретная СУБД).
- Минимальная поддерживаемая версия выбранной конкретной СУБД.
- Инфраструктурные особенности выбранной конкретной СУБД.

В этом списке пункты приведены в порядке убываниях их «судьбоносности» (так, например, переход в середине проекта с реляционной СУБД на иерархическую фактически будет означать начало проекта с нуля; а вот перенос нескольких серверов в облако может пройти почти безболезненно).

Поскольку выбор конкретной СУБД и её версии очень сильно влияет на дальнейшие технические решения, с этими вопросами тоже стоит разобраться до того, как на даталогическом уровне будет сделано много работы. Если придётся переделывать модель под другую версию выбранной СУБД или даже под другую СУБД, гарантированно потребуется очень много рутинной работы, а также, возможно, и интеллектуальной — если придётся искать новые технологические решения взамен уже созданным.

Соглашения относительно БД принимать проще, т.к. они слабо зависят от внешних факторов, а потому, как правило, почти не меняются. Что, однако, не снижает их важности. К таким решениям относятся (конкретные примеры будут в следующем разделе{307}):

- Соглашения об именовании структур.
- Соглашения об оформлении SQL-кода.
- Соглашения о комментариях.
- Иные специфические соглашения, зависящие от проекта (например, соглашения об API[197] в виде представлений{340} и хранимых подпрограмм{363}).

Конечно, никто не застрахован от таких изменений проектной ситуации, которые вынудят нас пересмотреть любое из только что обозначенных соглашений, но в наших силах продумать эти соглашения настолько хорошо, чтобы в будущем их не пришлось менять по нашей же собственной инициативе.

С инструментами на даталогическом уровне дела обстоят неоднозначно.

Для инфологического уровня нам в общем случае было достаточно любого текстового редактора (или любого удобного нам и представителям заказчика специализированного редактора UML (или иного графического языка) — таких инструментов много, и большинство специалистов хорошо умеет ими пользоваться).

Для даталогического уровня нам нужны специализированные инструменты, позволяющие оптимальным образом формировать структуру базы данных, многократно её пересматривать и анализировать, а в конечном итоге и экспортировать в полноценный SQL-код для импорта в СУБД.

Каждый производитель СУБД традиционно предоставляет свои собственные инструменты, например:

- MySQL Workbench[198] для MySQL.
- SQL Server Management Studio[199] для MS SQL Server.
- SQL Developer Data Modeler[200] для Oracle.
- И так далее — таких инструментов очень много.

---

197 **API** (application programming interface) — a computing interface which defines interactions between multiple software intermediaries. («Wikipedia») [https://en.wikipedia.org/wiki/Application_programming_interface]

198 «MySQL Workbench» [https://www.mysql.com/products/workbench/]

199 «SQL Server Management Studio» [https://docs.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms]

200 «SQL Developer Data Modeler» [https://www.oracle.com/database/technologies/appdev/datamodeler.html]

Это, безусловно, очень мощные инструменты, созданные с учётом всех особенностей целевой СУБД, и они однозначно заслуживают внимания. Если бы не одно «но»: они больше пригодятся вам для проектирования на следующем, физическом уровне{314}.

У каждого из перечисленных инструментов долгая история развития, их создатели ставили во главу угла разные цели, и в итоге каждый такой инструмент хорош в чём-то своём, но едва ли может претендовать на звание «эталонного решения».

На такое звание могут претендовать инструменты, изначально созданные для проектирования «баз данных в принципе», например:

- Sparx Systems Enterprise Architect[201] (большинство иллюстраций со схемами баз данных в данной книге сделаны именно с использованием этого инструмента).
- DbSchema[202].
- DbDiagram.io[203].
- И так далее — таких инструментов очень много.

Почему эти «универсальные» инструменты выигрывают? Вот лишь несколько самых очевидных преимуществ:

- Ни один из них не работает напрямую с физической базой данных (что сводит к нулю ваши шансы однажды уничтожить уже работающую базу данных заказчика).
- Эти инструменты поддерживают синтаксис и особенности многих СУБД, что позволяет вам сравнивать принимаемые решения для разных СУБД, переходить (конвертировать модели) с одной СУБД на другую или даже работать в таких необычных ситуациях, когда часть базы данных работает под управлением одной СУБД, а часть — под управлением другой СУБД.
- Создатели этих инструментов учли многие недостатки «узкоспециализированных» аналогов и постарались их исправить.
- Такие инструменты, как правило, обладают гораздо более дружественным интерфейсом, более развитыми средствами совместной (командной) работы и многими иными конкурентными преимуществами.

Естественно, вы имеете полное право выбора как между этими группами инструментов, так и между конкретными продуктами. А на стадии обучения рекомендуется попробовать поработать с хотя бы 3-5 принципиально различными инструментами, чтобы на личном опыте понять их различия и сформировать собственные предпочтения.

Задание 4.2.b: создайте даталогическую схему базы данных «Банк»{408} с использованием MySQL Workbench.

201 «Sparx Systems Enterprise Architect» [https://sparxsystems.com/products/ea/index.html]

202 «DbSchema» [https://dbschema.com]

203 «DbDiagram.io» [https://dbdiagram.io]

4.2.3. ПРИМЕР ПРОЕКТИРОВАНИЯ НА ДАТАЛОГИЧЕСКОМ УРОВНЕ

Как и было отмечено ранее, приступая к проектированию БД на даталогическом уровне, стоит принять ряд соглашений{304}. Сделаем это.

Соглашения относительно СУБД:
- Вид СУБД: реляционная.
- Конкретный продукт (конкретная СУБД): MySQL.
- Минимальная поддерживаемая версия выбранной конкретной СУБД: 8.0 и новее (т.к. более ранние версии имеют ряд технических ограничений).
- Инфраструктурные особенности выбранной конкретной СУБД: отдельный сервер[204].

Соглашения относительно БД:
- Соглашения об именовании структур:
  - При именовании таблиц и полей используется только нижний регистр.
  - Слова в именах таблиц и полей отделены друг от друга символом «_».
  - В именах таблиц существительные используются в единственном числе (например, «file», а не «files»). В именах полей множественное число допускается, но не рекомендуется.
  - Имена полей начинаются с префиксов из первых букв имён таблицы.
  - Имена ограничений начинаются с префикса «UNQ_» и содержат имена всех входящий в ограничение полей.
  - Имена триггеров начинаются с префикса «TRG_» и содержат в себе имя таблицы и название момента срабатывания.
- Соглашения об оформлении SQL-кода:
  - Все ключевые слова языка SQL пишутся в верхнем регистре.
  - Все имена структур БД обязательно должны быть заключены в обратные апострофы, т.е. символы «`».
- Соглашения о комментариях:
  - Комментарии оформляются для всех структур БД.
- Иные специфические соглашения, зависящие от проекта:
  - Все поля, содержащие дату и время, имеют тип `INTEGER` и хранят UNIXTIME-значения.
  - Все поля, содержащие только дату, имеют тип `DATE`.
  - Все первичные ключи — искусственные, автоинкрементируемые, беззнаковые.

В качестве основного инструмента проектирования мы будем использовать Sparx Enterprise Architect{306}, однако подчеркнём, что в особо сложных случаях и/или при проектировании большой БД может существовать промежуточный этап, на котором мы по-прежнему можем опираться на текстовое описание модели.

В случае с нашим конкретным учебным проектом явной необходимости в таком промежуточном этапе нет, но для полноты картины приведём соответствующий пример. Как правило, здесь будет использоваться табличное представление, т.к. для восприятия структурированной информации оно оказывается удобнее списков.

В процессе формирования этого примера мы также внесём некоторые правки в модель, уточнив то, что мы «забыли» на инфологическом уровне (см. задание 4.2.d{313}).

---

[204] В реальности, скорее всего, мы бы сразу ориентировались на кластер серверов, т.к. подобный проект предполагает достаточно высокую нагрузку, с которой отдельный сервер едва ли справится. Но из соображений упрощения учебного материала остановимся на боле тривиальном решении.

Таблица БД	Поле таблицы БД	Тип данных	Комментарии
Страница	Идентификатор	SMALLINT	Первичный ключ
	Родительская страница (rFK)	SMALLINT	
	Название страницы (для отображения на самой странице)	VARCHAR(500)	
	Имя страницы в меню (уникальное в рамках одного уровня меню)	VARCHAR(100)	Создать контролирующий триггер
	Заглавие страницы (HTML-тег TITLE)	VARCHAR(500)	
	Ключевые слова (HTML-тег meta … keywords)	VARCHAR(500)	
	Описание страницы (HTML-тег meta … description)	VARCHAR(500)	
	Текстовое наполнение страницы	TEXT	
Пользователь	Идентификатор	BIGINT	Первичный ключ
	Логин (уникальный)	VARCHAR(100)	Уникальное значение
	Пароль (sha256-хэш)	CHAR(64)	
	E-mail (уникальный)	VARCHAR(150)	Уникальное значение
	Дата и время регистрации (с точностью до секунды)	INT	
	Дата рождения (с точностью до дня)	DATE	
	Бонус в виде скорости по количеству закачанного (FK)	SMALLINT	
	Срок действия бонуса в виде скорости по количеству закачанного (до какой даты и времени, с точностью до секунды)	INT	
	Бонус в виде объёма по количеству закачанного (FK)	SMALLINT	
	Срок действия бонуса в виде объёма по количеству закачанного (до какой даты и времени, с точностью до секунды)	INT	
	Сколько файлов пользователь закачал	BIGINT	Агрегирующее поле, обновляется триггером
	Сколько файлов пользователь скачал	BIGINT	Агрегирующее поле, обновляется триггером
	Сколько комментариев оставил пользователь	BIGINT	Агрегирующее поле, обновляется триггером
	Сколько оценок оставил пользователь	BIGINT	Агрегирующее поле, обновляется триггером
	Причина блокировки (FK)	SMALLINT	
	До какой даты-времени (с точностью до секунды) действует блокировка	INT	

Таблица БД	Поле таблицы БД	Тип данных	Комментарии
Бонус	Идентификатор	SMALLINT	Первичный ключ
	За какое количество закачанных файлов выдаётся	BIGINT	
	За какой объём закачанных файлов (в байтах) выдаётся	BIGINT	
	Сколько добавляет к скорости скачивания (байт в секунду)	BIGINT	
	Сколько добавляет к объёму скачивания (в байтах)	BIGINT	
Группа пользователей	Идентификатор	SMALLINT	Первичный ключ
	Владелец (создатель) группы (FK)	BIGINT	
	Название группы (уникальное)	VARCHAR(100)	Уникальное значение
	Описание группы	TEXT	
Роль	Идентификатор	SMALLINT	Первичный ключ
	Название роли (уникальное)	VARCHAR(100)	Уникальное значение
	Ограничения по объёму закачиваемых файлов (в байтах)	BIGINT	NULL, если ограничения нет
	Ограничения по объёму скачиваемых файлов (в байтах)	BIGINT	NULL, если ограничения нет
	Ограничения по количеству закачиваемых файлов	BIGINT	NULL, если ограничения нет
	Ограничения по количеству скачиваемых файлов	BIGINT	NULL, если ограничения нет
	Ограничения по скорости закачивания (в байтах в секунду)	BIGINT	NULL, если ограничения нет
	Ограничения по скорости скачивания (в байтах в секунду)	BIGINT	NULL, если ограничения нет
Право	Идентификатор	SMALLINT	Первичный ключ
	Название права (уникальное)	VARCHAR(100)	Уникальное значение
	Описание права	TEXT	
Файл	Идентификатор	BIGINT	Первичный ключ
	Владелец	BIGINT	
	Размер (в байтах)	BIGINT	
	Дата и время добавления (с точностью до секунды)	INT	
	Срок хранения (до какой даты и какого времени, с точностью до секунды)	INT	NULL, если хранить бессрочно
	Исходное имя (без расширения)	VARCHAR(1000)	
	Исходное расширение	VARCHAR(1000)	
	Имя на сервере (sha256-хэш)	CHAR(64)	Уникальное значение
	Контрольная сумма (sha256-хэш)	CHAR(64)	
	Ссылка на удаление (для незарегистрированных пользователей, sha256-хэш)	CHAR(64)	Уникальное значение

Таблица БД	Поле таблицы БД	Тип данных	Комментарии
Ссылка на «публичное» скачивание файла	Идентификатор	BIGINT	Первичный ключ
	К какому файлу (FK)	BIGINT	
	Значение ссылки (sha256-хэш)	CHAR(64)	Уникальное значение
	Срок действия ссылки (дата и время, после которых ссылка считается недействительной и должна быть удалена, с точностью до секунды)	INT	NULL, если ссылка бессрочная
	Пароль на скачивание (если есть, sha256-хэш)	CHAR(64)	NULL, если пароль не требуется
Параметр доступа к файлу	Идентификатор	BIGINT	Первичный ключ
	К какому файлу (FK)	BIGINT	
	Какое действие (FK)	SMALLINT	
	Какому пользователю разрешено (FK)	BIGINT	Из этих двух полей одно обязательно должно быть NULL, другое — не NULL
	Какой группе разрешено (FK)	SMALLINT	
	Признак «разрешено всем зарегистрированным пользователям»	BIT(1)	
	Признак «разрешено вообще всем»	BIT(1)	
Оценка	Идентификатор	BIGINT	Первичный ключ
	Какой файл оценивается (FK)	BIGINT	
	Какой пользователь оценивает (FK)	BIGINT	
	Значение оценки (от 1 до 10)	TINYINT	
Комментарий	Идентификатор	BIGINT	Первичный ключ
	К какому файлу (FK)	BIGINT	Из этих двух полей одно обязательно должно быть NULL, другое — не NULL
	К какому комментарию (FK)	BIGINT	
	Текст комментария	TEXT	
	Какой пользователь оставил комментарий (FK)	BIGINT	
	Дата и время добавления комментария (с точностью до секунды)	INT	
	Выставленная оценка (FK)	BIGINT	Комментируя файл, можно не только писать текст, но и выставлять файлу оценку, оценку можно ставить только в комментариях верхнего уровня
Категория файлов	Идентификатор	SMALLINT	Первичный ключ
	Название категории (уникальное)	VARCHAR(100)	Уникальное значение
	Возрастные ограничения (если есть, FK)	SMALLINT	
Возрастное ограничение	Идентификатор	SMALLINT	Первичный ключ
	Минимальный возраст (в годах)	TINYINT	
	Название ограничения (уникальное)	VARCHAR(100)	Уникальное значение
	Описание ограничения	TEXT	

Таблица БД	Поле таблицы БД	Тип данных	Комментарии
Протокол	Пользователь (FK, NULL для незарегистрированных)	BIGINT	
	Ip-адрес	VARCHAR(45)	
	Операция (FK)	SMALLINT	
	Файл (если задействован, FK)	BIGINT	NULL, если операция не связана с файлами
	Дата и время выполнения действия (с точностью до секунды)	INT	
	Параметры действия (если есть)	TEXT	Сериализованный массив
Операция	Идентификатор	SMALLINT	Первичный ключ
	Название (уникальное)	VARCHAR(100)	Уникальное значение
Архив протокола (для записей старше месяца)	Пользователь (FK, NULL для незарегистрированных).	BIGINT	
	Ip-адрес	VARCHAR(45)	
	Операция (FK)	SMALLINT	
	Файл (если задействован, FK)	BIGINT	NULL, если операция не связана с файлами
	Дата и время выполнения действия (с точностью до секунды)	INT	
	Параметры действия (если есть)	TEXT	Сериализованный массив
Причина блокировки	Идентификатор	SMALLINT	Первичный ключ
	Название причины (уникальное)	VARCHAR(100)	Уникальное значение
	Описание причины	TEXT	
Чёрный список IP-адресов	IP-адрес в формате IPv4 (значение уникально)	CHAR(15)	Эти два поля не могут одновременно быть NULL (сделать триггер для проверки)
	IP-адрес в формате IPv6 (значение уникально)	CHAR(45)	
	Дата и время, до которых действительна блокировка (с точностью до секунды)	INT	
	Причина блокировки (FK)	SMALLINT	
Статистика	Всего зарегистрированных пользователей	BIGINT	Все поля этой таблицы являются агрегирующими и обновляются триггерами на соответствующих таблицах
	Пользователей зарегистрировалось сегодня	BIGINT	
	Всего закачано файлов	BIGINT	
	Файлов закачано сегодня	BIGINT	
	Общий объём закачанных файлов (в байтах)	BIGINT	
	Объём файлов, закачанных сегодня (в байтах)	BIGINT	
	Всего скачано файлов	BIGINT	
	Файлов скачано сегодня	BIGINT	
	Общий объём скачанных файлов (в байтах)	BIGINT	
	Объём файлов, скачанных сегодня (в байтах)	BIGINT	

Несмотря на то, что вышеприведённая таблица не очень сильно отличается от рассмотренного на инфологическом уровне списка сущностей и атрибутов{297}, она всё же содержит ряд важных дополнительных подробностей и может стать хорошим дополнением к проектной документации.

Теперь представим получившуюся модель графически (см. рисунок 4.2.а).

Здесь уже более двух десятков таблиц и большое количество связей между ними, да — модель получилась достаточно объёмной, а потому «на одной картинке» она сложно читается, но всё равно выглядит намного компактнее в сравнении с четырьмя страницами текстового описания. Особенно — если открыть исходный файл (см. раздаточный материал к книге{5}) в Sparx Enterprise Architect и выбрать удобный лично вам масштаб отображения.

Можно заметить, что на рисунке 4.2.а таблиц больше, чем было представлено ранее в текстовом описании{308}. Это связано с тем, что в процессе формирования модели были добавлены таблицы для связей «многие ко многим{60}», а также внесены иные небольшие правки (в для т.ч. адаптации к требованию удобства использования БД{14}).

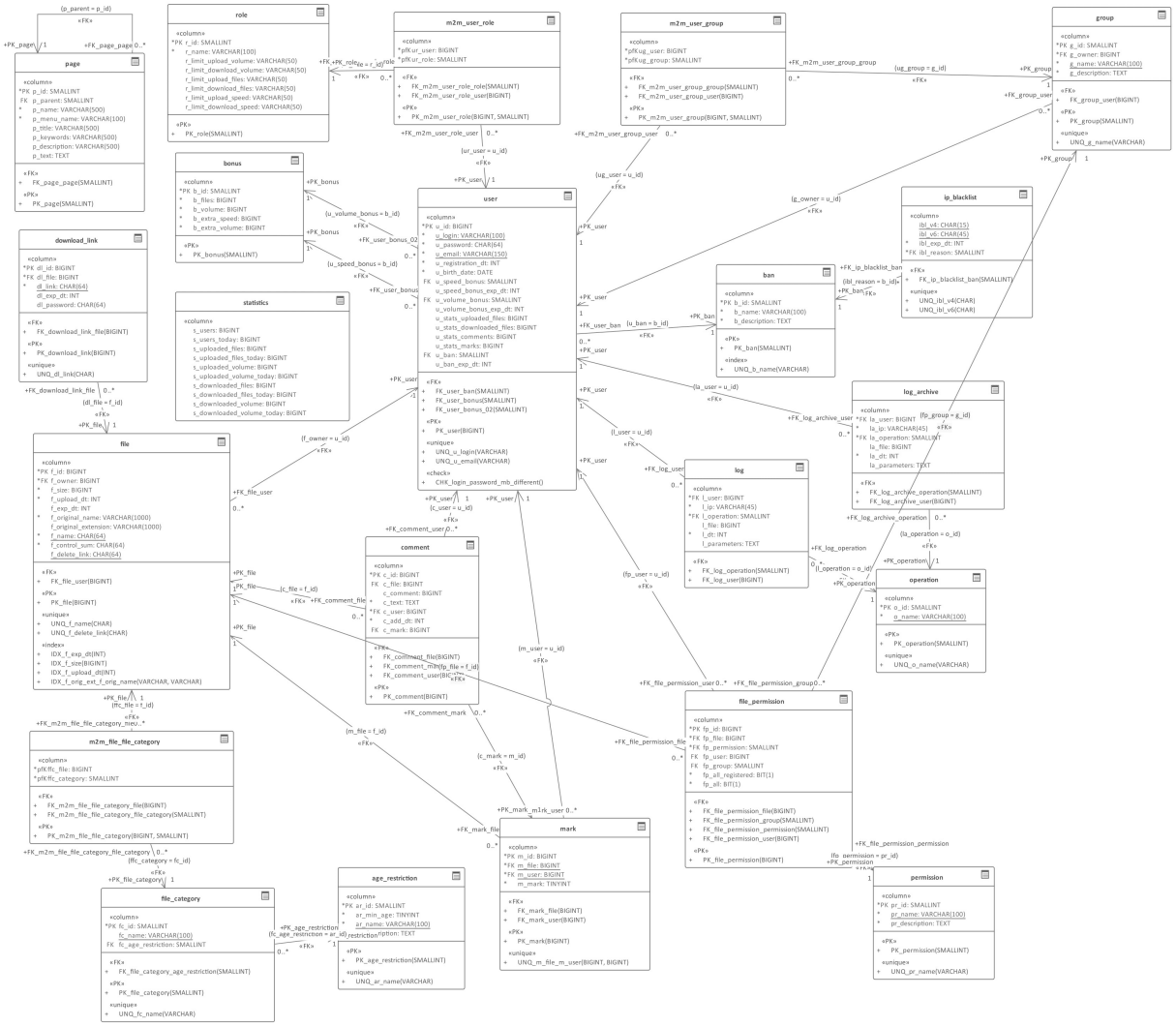


*Рисунок 4.2.а — **Графическое представление результатов проектирования на даталогическом уровне***[205]

---

205 Полноразмерную картинку вы можете найти в раздаточном материале{5} или по этой ссылке: http://svyatoslav.biz/relational_databases_book_download/pics/logical_level_graphical_representaion.png.

Также в процессе создания этой модели были добавлены почти все необходимые ограничения уникальности{109}, проведены связи между таблицами, и все комментарии были «вписаны» в модель (для того, чтобы в будущем при генерации SQL-кода они были перенесены в реальную базу данных).

Уже начиная с этого момента, мы можем (и даже должны) периодически генерировать SQL-код и импортировать получившийся результат в СУБД, чтобы избежать различных досадных ошибок, совершать которые склонен любой человек (если с моделью «что-то не так», импорт завершится неудачей, и СУБД подробно опишет причину в сообщениях об ошибках). Намного проще исправить подобные недочёты сейчас, пока модель ещё достаточно «сырая», и её доработка не потребует больших усилий.

Так, например, при подготовке материала данной книги первая попытка переноса модели в СУБД завершилась следующей ошибкой:

MySQL **Сообщение СУБД об ошибке выполнения SQL-скрипта**

```
1   ALTER TABLE `page` ADD CONSTRAINT `fk_page_page` FOREIGN KEY (`p_parent`)
2   REFERENCES `page` (`p_id`)
3   ON DELETE RESTRICT
4   ON UPDATE RESTRICT
5   Error Code: 1825. Failed to add the foreign key constraint on table 'page'.
6   Incorrect options in FOREIGN KEY constraint 'FK_page_page'.
```

И действительно, первичный ключ таблицы **`page`** был беззнаковым числом, в то время как рекурсивный внешний ключ этой же таблицы был обычным числом (способным принимать отрицательные значения).

После исправления этой (и пары аналогичных ей) ошибок импорт в СУБД прошёл успешно.

И тем не менее, до завершения проектирования остаётся ещё один очень насыщенный этап — создание модели базы данных на физическом уровне.

Задание 4.2.c: доработайте даталогическую схему базы данных «Банк»{408} так, чтобы устранить все обнаруженные в ней недостатки (насколько это возможно без привлечения «гипотетического заказчика» для получения ответов на возникающие вопросы).

Задание 4.2.d: ранее{307} в данной главе было отмечено, что в процессе промежуточного этапа проектирования (на котором мы по-прежнему опирались на текстовое описание модели) мы вносили некоторые правки, которые «забыли» на инфологическом уровне. Какие правки мы сделали? Составьте полный список.

Задание 4.2.e: какие ещё данные вы бы добавили в итоговую таблицу{308}, отражающую итог даталогического проектирования базы данных файлообменного сервиса? Внесите соответствующие правки.

4.3. ПРОЕКТИРОВАНИЕ НА ФИЗИЧЕСКОМ УРОВНЕ

4.3.1. ЦЕЛИ И ЗАДАЧИ ПРОЕКТИРОВАНИЯ НА ФИЗИЧЕСКОМ УРОВНЕ

Перед прочтением материала данной главы стоит повторить основы моделирования баз данных{8}.

Начиная с этого момента, мы вступаем в область, в которой очень мало универсальных решений, поскольку, как следует из определения физического уровня моделирования{11}, мы будем вынуждены максимально учитывать технические особенности и возможности конкретной СУБД.

И всё же некоторые общие цели и задачи можно выделить даже здесь, т.к. особый интерес для нас представляют:

- права доступа;
- кодировки;
- методы доступа;
- индексы;
- настройки СУБД.

Права доступа

В относительно небольших и простых проектах предполагается, что взаимодействие с СУБД идёт от имени одного пользователя — работающее с СУБД приложение всегда авторизуется с использованием одной и той же пары «логин-пароль», и само контролирует права своих пользователей (см. рисунок 4.3.а).

*Рисунок 4.3.a — **Работа с СУБД от имени одного пользователя***

Этот вариант получил широкое распространение из-за своей простоты и скорости реализации. Но если мы рассмотрим ситуацию, когда с одной и той же базой данных может работать много разных приложений, а к безопасности предъявляются повышенные требования, имеет смысл переложить аутентификацию и контроль прав доступа на СУБД (см. рисунок 4.3.b).

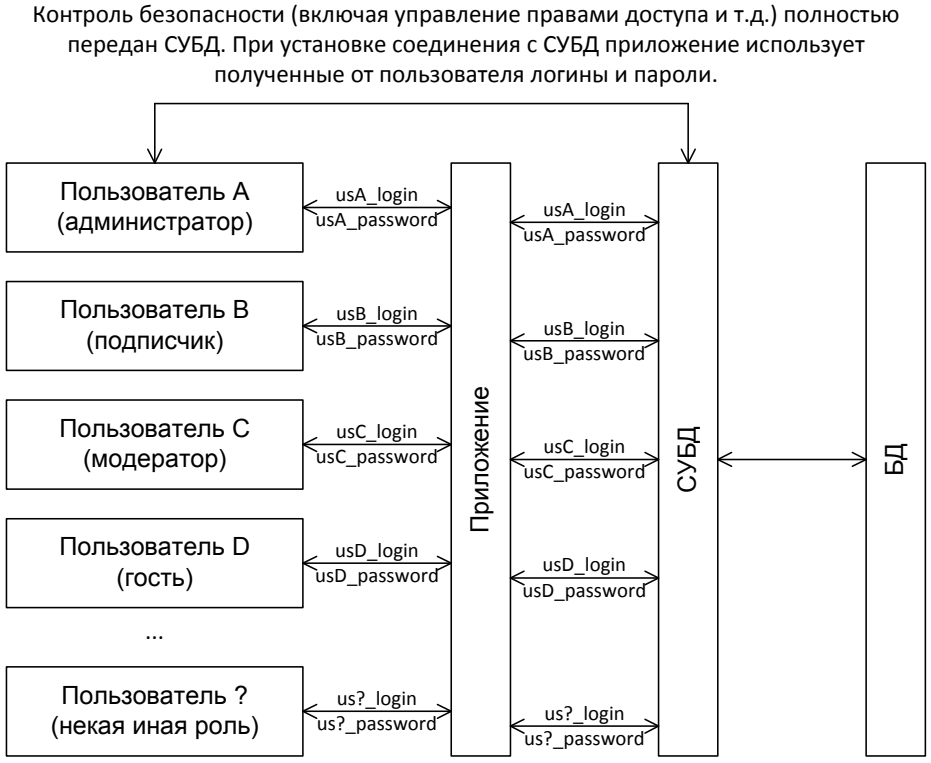


*Рисунок 4.3.b — **Работа с СУБД от имени нескольких пользователей***

Да, такой подход намного более сложен в реализации, он требует не только постоянного администрирования СУБД (как минимум в плане управления имеющимися пользователями и их правами), но также требует серьёзных доработок в самой базе данных (например, для доступа к данным может быть создан дополнительный уровень абстракции в виде представлений{340} и хранимых процедур{363} — такой подход позволяет намного более гибко управлять правами доступа и защищать данные).

Но несмотря на свою ощутимо большую сложность, такой подход позволяет:

- обеспечить намного более высокий уровень безопасности;
- исключить необходимость дублирования модели безопасности во множестве отдельных приложений (модель один раз реализуется на уровне БД/СУБД и распространяется на всех клиентов);
- исключить ситуации осознанного или случайного «обхода» системы безопасности (когда, например, работа идёт напрямую с базой данных в процессе технического обслуживания или устранения неполадок).

Кодировки

Одной из самых частых ошибок начинающих при проектировании баз данных является недостаточное внимание кодировкам символов.

Ситуация многократно усугубляется тем, что русскоговорящие разработчики часто используют русифицированное программное обеспечение[206] (включая операционную систему), что приводит к автоматической конфигурации СУБД таким образом, что на компьютере у разработчика «всё работает». А при переносе на полноценный сервер — перестаёт.

Если при проектировании СУБД не указать явным образом кодировки базы данных, таблиц (а в некоторых СУБД — даже отдельных полей таблиц), то при переносе модели базы данных в СУБД все кодировки будут выставлены «по умолчанию», т.е. взяты из настроек той СУБД, в которую импортируется база данных.

Продемонстрируем на простом примере, к чему это приводит.

Предположим, что мы используем MySQL, и у разработчика на его компьютере кодировкой по умолчанию является `utf8`. При этом на одном из серверов, на которых будет работать создаваемая база данных, кодировкой по умолчанию является `latin1`.

Для наглядности запредельно упростим пример и создадим всего лишь одну таблицу с одним текстовым полем:

MySQL **Ошибочное создание таблицы (не указаны кодировки)**

```
1   CREATE TABLE `words`
2   (
3     `word` VARCHAR(255) NULL
4   )
```

Наполним эту таблицу данными (вернее — безуспешно попытаемся):

MySQL **Ошибка при добавлении русского текста в UTF8**

```
1   INSERT INTO `words`
2               (`word`)
3   VALUES      ('Груша'),
4               ('Яблоко'),
5               ('Апельсин')
```

Уже на этом этапе мы получаем сообщение об ошибке:

MySQL **Сообщение об ошибке**

```
1   Error Code: 1366. Incorrect string value:
2   '\xD0\x93\xD1\x80\xD1\x83...' for column 'word' at row 1
```

---

206 Пожалуйста, никогда так не делайте. Если вы занимаетесь разработкой программного обеспечения и/или баз данных, всё программное обеспечение на вашем компьютере должно быть на английском языке. Без исключений.

При этом можно смело утверждать, что нам ещё очень повезло. При иных стечениях обстоятельств могла возникнуть ситуация, в которой вставка данных проходит успешно, а вот упорядочивание и поиск «не работают», т.е. выдают неправильные результаты.

А всего-то нужно было не полениться и указать в своём инструменте проектирования кодировку символов, чтобы запрос на создание таблицы выглядел так (обратите внимание на строки 5 и 6 запроса):

MySQL **Правильное создание таблицы (кодировки указаны)**

```
1   CREATE TABLE `words`
2   (
3    `word` VARCHAR(255) NULL
4   )
5   DEFAULT CHARACTER SET = utf8
6   COLLATE = utf8_general_ci
```

Теперь запросы на добавление, упорядочивание, поиск и т.д. будут выдавать правильные ожидаемые результаты.

Да, эту проблему очень легко заметить. Но если вы создаёте приложение, многими копиями которого будет пользоваться большое количество ваших клиентов (и вы не можете заранее знать настройки их СУБД в каждом конкретном случае), готовьтесь к шквалу гневных сообщений о том, что «ничего не работает». Или настраивайте кодировки везде, где это только возможно.

Отдельно обращаюсь к тем, у кого «всегда всё работало и без этого». Нет. Оно не работало, оно «срабатывало». Т.е. до сих пор вам просто везло. Теперь вы знаете, как повысить надёжность разрабатываемых вами решений, а как будете делать именно вы — решать, конечно же, вам.

Методы доступа

Методы доступа уже были упомянуты ранее{33}, и здесь мы лишь отметим, что к ним в полной мере относятся все проблемы и рекомендации, только что описанные на примере кодировок.

Если полагаться на настройки по умолчанию, легко оказаться в ситуации, когда поведение СУБД будет совсем не таким, какое вы ожидали. Могут возникнуть проблемы с производительностью, контролем ссылочной целостности (да, до сих пор существуют методы доступа, не поддерживающие такой контроль[207]), транзакциями, блокировками таблиц при операциях модификации, репликациями, масштабированием и т.д. и т.п.

Метод доступа всецело определяет поведение базы данных на самом низком уровне — на уровне файлов, в которых хранятся данные. Потому крайне неосмотрительно было бы полагаться на некие умолчания. Напротив — всегда стоит указывать метод доступа явно, если выбранная вами СУБД это позволяет.

Например, в MySQL за это отвечает параметр **`ENGINE`** в синтаксисе создания таблицы. Вы лишь должны явно указать его с использованием выбранного вами инструмента проектирования так, чтобы эта информация попала в итоговый SQL-запрос, с помощью которого будет создаваться ваша база данных (см. строку 5 в приведённом ниже примере запроса):

MySQL **Правильное создание таблицы (указан метод доступа)**

```
1   CREATE TABLE `words`
2   (
3    `word` VARCHAR(255) NULL
4   )
5   ENGINE = InnoDB
6   DEFAULT CHARACTER SET = utf8
7   COLLATE = utf8_general_ci
```

207 Например, метод доступа MyISAM в MySQL.

Индексы

Индексам уже был посвящён обширный раздел данной книги{106}. Из представленной там информации следует, что существует широчайший выбор вариантов как разновидностей индексов, так и их параметров.

Многие индексы (например, уникальные{109} или на полях, по которым очевидно очень часто будет выполняться поиск), как правило, видны ещё на даталогическом уровне моделирования — и там же создаются. Но для того и существует физический уровень, чтобы не только ещё раз проверить, не забыт ли какой-то из «очевидных» индексов, но и выполнить серию дополнительных действий:

- наполнить базу данных тестовыми данными, максимально приближенными к реальным (как по объёму, так и по содержанию);
- на основе уже имеющейся информации о типичных запросах провести нагрузочное тестирование{397};
- определить те узкие места{397} в производительности базы данных, которые можно устранить с использованием индексов;
- создать необходимые индексы;
- сделать пометку на будущее о необходимости периодической проверки эффективности созданных индексов как в процессе разработки базы данных и работающих с ней приложений, так и в процессе реальной работы созданного проекта.

Создание и удаление индексов (за исключением уникальных{109} и первичных{110}) влияет не на структуру и логику работы базы данных, а на её производительность. Поэтому вносить соответствующие правки можно смело и в любой момент времени (хоть и предполагается, что при правильном проектировании этот момент наступит раньше, чем с базой данных начнут работать реальные пользователи).

Основная же сложность состоит в том, что при тонкой настройке индексов необходимо не только проводить много экспериментов, но и тщательно изучать техническую документацию к конкретной версии конкретной СУБД, т.к. очень часто знакомые вам по одной ситуации решения ведут себя совершенно иначе в другой ситуации.

Настройки СУБД

Здесь мы достигли предела в невозможности сформулировать хоть какие-нибудь общие рекомендации. Вопрос «как настроить СУБД?» без указания сотен дополнительных деталей звучит так же абстрактно и нелепо, как и, например, вопрос «как приготовить еду?» или «как нарисовать картину?».

Самый честный совет, который можно дать начинающим разработчикам баз данных, звучит так: не меняйте никакие настройки СУБД, если вы не понимаете совершенно чётко, что именно, почему, зачем и как вы делаете. В большинстве случаев настройки СУБД по умолчанию «оптимизированы» под широкий спектр типичных решений, к которым, скорее всего, и относится ваша база данных.

Если же создаваемый вами продукт выходит за рамки «типичных», то вот эти его отличительные особенности вы и должны учитывать, изучая документацию и экспериментируя с настройками СУБД.

И обязательно помните о двух вещах:

- создавайте резервные копии перед каждым внесением изменений;
- многократно всё проверьте на тестовом окружении перед тем, как вносить правки в настройки СУБД, с которой уже работают реальные пользователи.

Задание 4.3.а: сформулируйте список вопросов, ответы на которые помогли бы вам улучшить физический уровень моделирования базы данных «Банк»{412}.

4.3.2. ИНСТРУМЕНТЫ И ТЕХНИКИ ПРОЕКТИРОВАНИЯ НА ФИЗИЧЕСКОМ УРОВНЕ

В предыдущей главе было отмечено, что на данном уровне моделирования нас интересуют права доступа, кодировки, методы доступа, индексы, настройки СУБД.

Что касается техник и подходов к принятию необходимых решений, то ничего принципиально нового здесь не добавляется — мы всё также должны собирать информацию от:

- заказчика — чтобы максимально полно понять специфику проекта, границы бюджета (это может ощутимо повлиять на выбор доступных нам конфигураций СУБД), типичные сценарии работы с базой данных и т.д.;
- разработчиков взаимодействующего с базой данных приложения (или приложений, если их несколько) — чтобы более точно спрогнозировать типичную форму нагрузки на базу данных, увидеть специфику выполняемых запросов, оценить требования к безопасности и т.д.;
- технических специалистов (если это — не мы) дата-центра (или облачного сервиса), в котором будет расположена СУБД с нашей базой данных — чтобы заблаговременно знать все ключевые ограничения и доступные нам возможности по настройке СУБД и инфраструктуры.

Поскольку интересующая нас информация крайне разнообразна по составу и форме представления, мы можем использовать для её организации и интеграции в модель базы данных любой из рассмотренных ранее в данном разделе инструментов.

Как минимум, управление кодировками, методами доступа и индексами доступно практически в любом инструменте моделирования на даталогическом уровне{304}. Причём те инструменты, которые на даталогическом уровне были отмечены как излишне узкоспециализированные{305}, здесь, напротив, могут оказаться наиболее полезными.

Что касается управления правами доступа и настройками СУБД, то самым выгодным инструментом здесь будет такой, который позволит вам автоматически настраивать эти параметры каждый раз при генерации базы данных и/или развёртывании её в СУБД. Т.е. такой инструмент должен уметь автоматически выполнять в нужный момент времени набор SQL-запросов, модифицировать текстовые файлы, вносить изменения в реестр Windows и т.д.

Всеми этими (и множеством других) возможностями обладают DevOps[208]-инструменты. Поскольку их количество огромно, а скорость их развития и изменения колоссальна, список конкретных рекомендованных инструментов будет устаревать буквально каждую неделю, но ничто не мешает вам в любой момент времени обратиться за рекомендациями к более опытным коллегам или элементарно «загуглить» наиболее оптимальный для вашей конкретной ситуации инструмент.

Несколько наглядных примеров будет представлено в следующей главе, а в завершение этой представим небольшую «шпаргалку», которая обобщает материал двух последних глав в краткой форме и также содержит общие советы (которые в наибольшей степени пригодятся начинающим).

---

208 **DevOps** — a set of practices that combines software development (Dev) and IT operations (Ops). It aims to shorten the systems development life cycle and provide continuous delivery with high software quality. («Wikipedia») [https://en.wikipedia.org/wiki/DevOps]

Что	Когда начинать обдумывать	Когда заканчивать реализовывать	Инструменты	Общий совет
Права доступа	На инфологическом уровне	К моменту ввода БД в эксплуатацию	SQL-скрипты + DevOps-инструменты	Если вы не создаёте по-настоящему сложный и масштабный проект, реализуйте представленный на рисунке 4.3.a{315} подход и сэкономьте силы и время
Кодировки	На инфологическом уровне	К моменту приёмочного тестирования	Средство проектирования, используемое на даталогическом уровне	Обязательно прописывайте кодировки явным образом везде, где это позволяет выбранная вами СУБД
Методы доступа	На даталогическом уровне	К моменту приёмочного тестирования	Средство проектирования, используемое на даталогическом уровне	Обязательно прописывайте методы доступа явным образом везде, где это позволяет выбранная вами СУБД
Индексы	На даталогическом уровне	Можно изменять даже в процессе эксплуатации СУБД	Средство проектирования, используемое на даталогическом уровне + SQL-скрипты + DevOps-инструменты + средства нагрузочного тестирования	Обязательно проводите исследования (в т.ч. нагрузочное тестирование), не полагайтесь на то, что некоторые решения будет просто казаться логичными — здесь может быть много сюрпризов
Настройки СУБД	На даталогическом уровне	Можно изменять даже в процессе эксплуатации СУБД	SQL-скрипты + DevOps-инструменты	Не меняйте никакие настройки СУБД, если вы не понимаете совершенно чётко, что именно, почему, зачем и как вы делаете

Итак, переходим к примерам.

Задание 4.3.b: доработайте с использованием Sparx Enterprise Architect модель базы данных «Банк»{408} так, чтобы отразить в ней все необходимые нюансы физического уровня проектирования (см. раздаточный материал{5}).

4.3.3. ПРИМЕР ПРОЕКТИРОВАНИЯ НА ФИЗИЧЕСКОМ УРОВНЕ

На физическом уровне довольно сложно визуализировать как процесс проектирования, так и его результат. Тем более, что наша учебная база данных часто не будет нуждаться в каких-то особых доработках. Но мы постараемся раскрыть весь процесс настолько полно, насколько это можно сделать в виде текста и картинок.

Права доступа

Да, в нашем конкретном случае намного рациональнее использовать работу с СУБД от имени одного пользователя (см. рисунок 4.3.a{315}), но если бы мы решили пойти по более сложному пути (см. рисунок 4.3.b{315}), реализовать это можно было бы следующим образом.

В первую очередь необходимо определить некие глобальные роли пользователей и перечень их прав относительно объектов базы данных.

Для краткости будем использовать общепринятые сокращения, произошедшие от известной аббревиатуры CRUD[209]: C — право на создание записи, R — право на чтение записи, U — право на обновление записи, D — право на удаление записи.

	Приложение	Гость	Пользователь	Модератор	Администратор
age_restriction	R	R	R	CRUD	CRUD
ban	R	R	R	R	CRUD
bonus	R	R	R	R	CRUD
comment	R	R	R	CRUD	CRUD
download_link	R	R	R	CRUD	CRUD
file	R	R	CRUD	CRUD	CRUD
file_category	R	R	R	CRUD	CRUD
file_permission	R	R	CRUD	CRUD	CRUD
group	R	R	R	CRUD	CRUD
ip_blacklist	CRUD	-	-	-	CRUD
log	C	-	-	-	CRUD
log_archive	C	-	-	-	CRUD
m2m_file_file_category	R	R	CRUD	CRUD	CRUD
m2m_user_group	R	R	R	CRUD	CRUD
m2m_user_role	R	R	R	CRUD	CRUD
mark	R	R	CRUD	CRUD	CRUD
operation	R	-	-	-	CRUD
page	R	R	R	CRUD	CRUD
permission	R	R	R	R	CRUD
role	R	R	R	R	CRUD
statistics	R	R	R	R	CRUD
user	CRUD	R	CRUD	CRUD	CRUD

209 CRUD — Create, Read, Update, Delete (создать, прочитать, обновить, удалить).

Очевидно, что часть операций (например, протоколирование) приложение должно выполнять в любом случае — вне зависимости от того, какой пользователь с ним сейчас работает — потому нам придётся создать отдельную роль (и отдельного пользователя) именно для самого приложения. От имени этого же пользователя будет происходить регистрация остальных пользователей.

Теперь необходимо создать соответствующий SQL-код. Для краткости приведём таковой для ролей «Приложение» и «Администратор» (создание аналогичного кода для ролей «Гость», «Пользователь» и «Модератор» оставим на задание для самостоятельной проработки 4.3.d{330}).

MySQL **Создание пользователей и управление их правами**

```
-- 1) Пользователь для роли "Приложение":
DROP USER IF EXISTS 'feapp'@'localhost';
CREATE USER 'feapp'@'localhost' IDENTIFIED BY '<сложный пароль>';

GRANT SELECT ON `age_restriction` TO 'feapp'@'localhost';
GRANT SELECT ON `ban` TO 'feapp'@'localhost';
GRANT SELECT ON `bonus` TO 'feapp'@'localhost';
GRANT SELECT ON `comment` TO 'feapp'@'localhost';
GRANT SELECT ON `download_link` TO 'feapp'@'localhost';
GRANT SELECT ON `file` TO 'feapp'@'localhost';
GRANT SELECT ON `file_category` TO 'feapp'@'localhost';
GRANT SELECT ON `file_permission` TO 'feapp'@'localhost';
GRANT SELECT ON `group` TO 'feapp'@'localhost';
GRANT INSERT, SELECT, UPDATE, DELETE ON `ip_blacklist`
                                     TO 'feapp'@'localhost';
GRANT SELECT ON `log` TO 'feapp'@'localhost';
GRANT SELECT ON `log_archive` TO 'feapp'@'localhost';
GRANT SELECT ON `m2m_file_file_category` TO 'feapp'@'localhost';
GRANT SELECT ON `m2m_user_group` TO 'feapp'@'localhost';
GRANT SELECT ON `m2m_user_role` TO 'feapp'@'localhost';
GRANT SELECT ON `mark` TO 'feapp'@'localhost';
GRANT SELECT ON `operation` TO 'feapp'@'localhost';
GRANT SELECT ON `page` TO 'feapp'@'localhost';
GRANT SELECT ON `permission` TO 'feapp'@'localhost';
GRANT SELECT ON `role` TO 'feapp'@'localhost';
GRANT SELECT ON `statistics` TO 'feapp'@'localhost';
GRANT INSERT, SELECT, UPDATE, DELETE ON `user` TO 'feapp'@'localhost';
GRANT CREATE USER ON *.* TO 'feapp'@'localhost' WITH GRANT OPTION;

-- 2) Пользователи для ролей "Гость", "Пользователь", "Модератор"
--    будут управляться аналогичным образом.

-- 3) Пользователь для роли "Администратор":
DROP USER IF EXISTS 'feadmin'@'localhost';
CREATE USER 'feadmin'@'localhost' IDENTIFIED BY '<сложный пароль>';

GRANT ALL PRIVILEGES ON * TO 'feadmin'@'localhost';
```

Если мы захотим использовать ещё более сложную систему контроля прав, мы можем создать представления{340} и хранимые процедуры{363}, выдать соответствующие права только на эти объекты, а прямой доступ к таблицам запретить вообще всем.

Кодировки

Логика управления кодировками очень сильно зависит от возможностей выбранного вами средства проектирования и от того, будут ли кодировки различными для разных объектов базы данных.

В предельном случае придётся прописать кодировки вручную для каждой таблицы. Если вы используете Sparx Enterprise Architect, это можно сделать через так называемые tagged values таблиц. Примерная последовательность действий показана на рисунке 4.3.c.

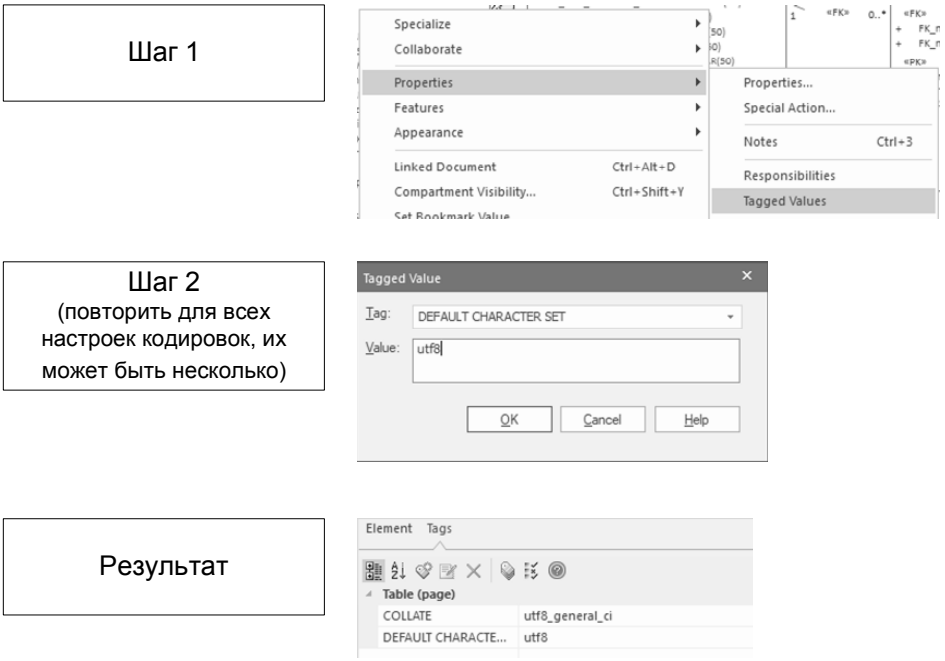


*Рисунок 4.3.c — **Установка кодировок с использованием tagged values***

Если же кодировки для всех объектов базы данных одинаковы — пожалуй, самый надёжный способ состоит в том, чтобы создать хранимую процедуру, управляющую кодировками всех интересующих нас объектов (как правило — это только таблицы) и выполнять её в процессе создания базы данных.

Код создания и вызова такой процедуры для MySQL может выглядеть подобным образом[210].

MySQL **Хранимая процедура для управления кодировками всех таблиц в базе данных**

```
1   DROP PROCEDURE IF EXISTS SET_ENCODING_TO_ALL_TABLES;
2
3   DELIMITER $$
4   CREATE PROCEDURE SET_ENCODING_TO_ALL_TABLES
5      (IN default_charset_name VARCHAR(150), IN collation_name VARCHAR(150))
6   BEGIN
7    DECLARE done INT DEFAULT 0;
8    DECLARE tbl_name VARCHAR(200) DEFAULT '';
9    DECLARE all_tables_cursor CURSOR FOR
10    SELECT `table_name`
11      FROM `information_schema`.`tables`
12      WHERE `table_schema` = DATABASE()
13        AND `table_type` = 'BASE TABLE';
14   DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
15
16   OPEN all_tables_cursor;
17   tables_loop: LOOP
18    FETCH all_tables_cursor INTO tbl_name;
19    IF done
20     THEN LEAVE tables_loop;
21    END IF;
22
23
24
25
```

210 Подробности о том, как работают таких хранимые процедуры, можно найти в разделе 5.2 «Использование хранимых процедур» книги «Работа с MySQL, MS SQL Server и Oracle в примерах» (С.С. Куликов) [http://svyatoslav.biz/database_book/]

```
26    SET @alter_table_query = CONCAT('ALTER TABLE `', tbl_name,
27        '` CONVERT TO CHARACTER SET \'', default_charset_name,
28        '\' COLLATE \'', collation_name, '\'');
29
30    PREPARE alter_table_stmt FROM @alter_table_query;
31    EXECUTE alter_table_stmt;
32    DEALLOCATE PREPARE alter_table_stmt;
33   END LOOP tables_loop;
34   CLOSE all_tables_cursor;
35  END;
36  $$
37  DELIMITER ;
38
39  CALL SET_ENCODING_TO_ALL_TABLES('utf8', 'utf8_general_ci');
```

В данном конкретном случае можно было обойтись и без хранимой процедуры (см. задание для самостоятельной проработки 4.3.e{330}), но так мы получаем хорошую заготовку на будущее, которую можно будет использовать с минимальными доработками и для решения более сложных задач.

Методы доступа

Управление методами доступа в точности идентично управлению кодировками. Мы можем либо настроить их заранее в средстве проектирования (см. на рисунке 4.3.d краткую инструкцию по выполнению этой операции в Sparx Enterprise Architect), либо использовать SQL-код.

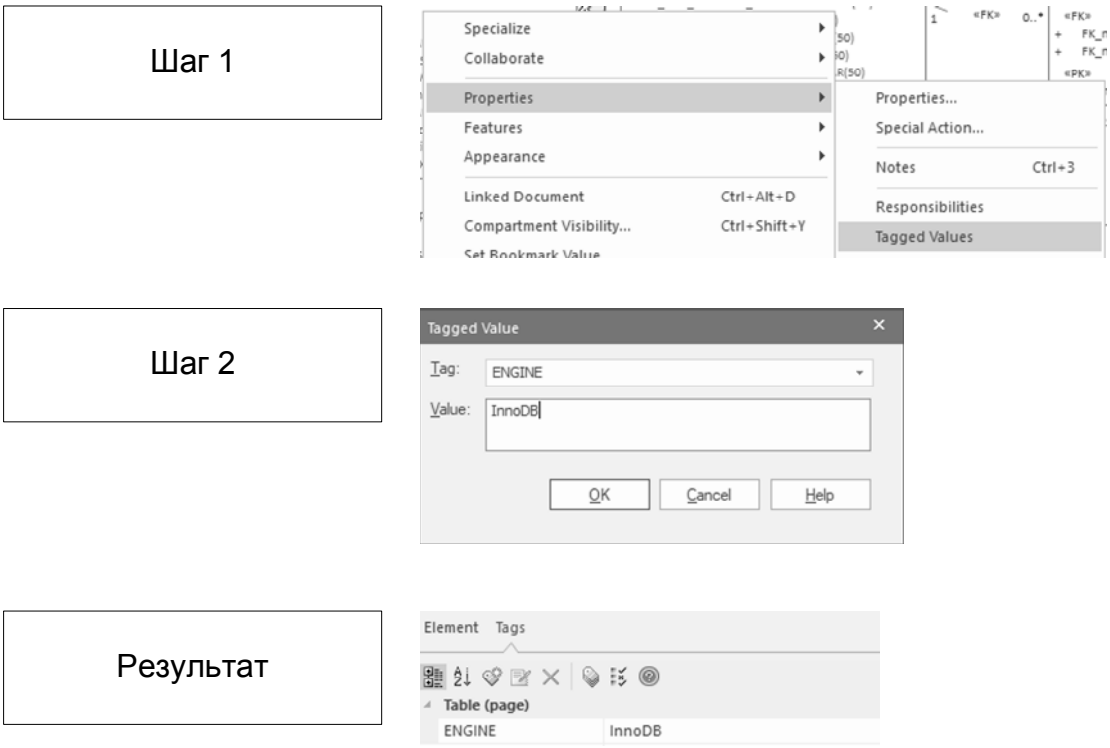


*Рисунок 4.3.d — **Выбор метода доступа с использованием tagged values***

Поскольку у нас уже есть почти готовая хранимая процедура, внесём в неё небольшие правки с тем, чтобы она устанавливала не только кодировки всех таблиц, но и методы доступа.

MySQL Доработанная хранимая процедура для управления кодировками и методами доступа всех таблиц

```
DROP PROCEDURE IF EXISTS SET_ENCODING_AND_STORAGE_ENGINE_TO_ALL_TABLES;

DELIMITER $$
CREATE PROCEDURE SET_ENCODING_AND_STORAGE_ENGINE_TO_ALL_TABLES
          (IN default_charset_name VARCHAR(150),
           IN collation_name VARCHAR(150),
           IN storage_engine VARCHAR(150))
BEGIN
 DECLARE done INT DEFAULT 0;
 DECLARE tbl_name VARCHAR(200) DEFAULT '';
 DECLARE all_tables_cursor CURSOR FOR
  SELECT `table_name`
    FROM `information_schema`.`tables`
    WHERE `table_schema` = DATABASE()
      AND `table_type` = 'BASE TABLE';
 DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

 OPEN all_tables_cursor;
 tables_loop: LOOP
  FETCH all_tables_cursor INTO tbl_name;
  IF done
   THEN LEAVE tables_loop;
  END IF;

  SET @alter_table_encoding_query = CONCAT('ALTER TABLE `', tbl_name,
    '` CONVERT TO CHARACTER SET \'', default_charset_name,
    '\' COLLATE \'', collation_name, '\'');
  SET @alter_table_engine_query = CONCAT('ALTER TABLE `', tbl_name,
    '` ENGINE = \'', storage_engine, '\'');

  PREPARE alter_table_encoding_stmt FROM @alter_table_encoding_query;
  PREPARE alter_table_engine_stmt FROM @alter_table_engine_query;

  EXECUTE alter_table_encoding_stmt;
  EXECUTE alter_table_engine_stmt;

  DEALLOCATE PREPARE alter_table_encoding_stmt;
  DEALLOCATE PREPARE alter_table_engine_stmt;
 END LOOP tables_loop;
 CLOSE all_tables_cursor;
END;
$$
DELIMITER ;

CALL SET_ENCODING_AND_STORAGE_ENGINE_TO_ALL_TABLES('utf8',
                                                   'utf8_general_ci', 'InnoDB');
```

Очевидно, что настройка методов доступа не сводится просто к выбору самого метода доступа — существует огромное количество дополнительных параметров, которые необходимо учитывать в каждой конкретной ситуации. Здесь единственный верный способ — внимательно читать документацию[211], проводить исследования, экспериментировать.

---

[211] Так, например, с документацией по настройке InnoDB для MySQL 8.0 можно ознакомиться здесь: https://dev.mysql.com/doc/refman/8.0/en/innodb-configuration.html.

Индексы

Как следует из соответствующего раздела данной книги{106}, существует огромное количество индексов, различающихся по целям и задачам, логике работы, физической организации и множеству иных параметров.

На даталогическом уровне мы уже создали самые очевидные индексы, отвечающие за уникальность значений некоторых полей и ускоряющие выполнение операции JOIN (они автоматически создаются на внешних ключах).

Поскольку создание и удаление индексов (за исключением уникальных{109} и первичных{110}) не влияет на структуру и логику работы базы данных, необходимые правки можно вносить в любой момент времени, но в подавляющем большинстве случаев требуется не только тщательный анализ ситуации, но и изучение планов выполнения запросов и проведение серии экспериментов.

Проиллюстрируем эту логику на одном примере (в остальных случаях действия будут теми же, отличия будут только в собранных данных).

Рассмотрим таблицу `file`. Она является одновременно как одной из наиболее объёмных в нашей базе данных, так и одной из наиболее нагруженных (по количеству обращений к ней).

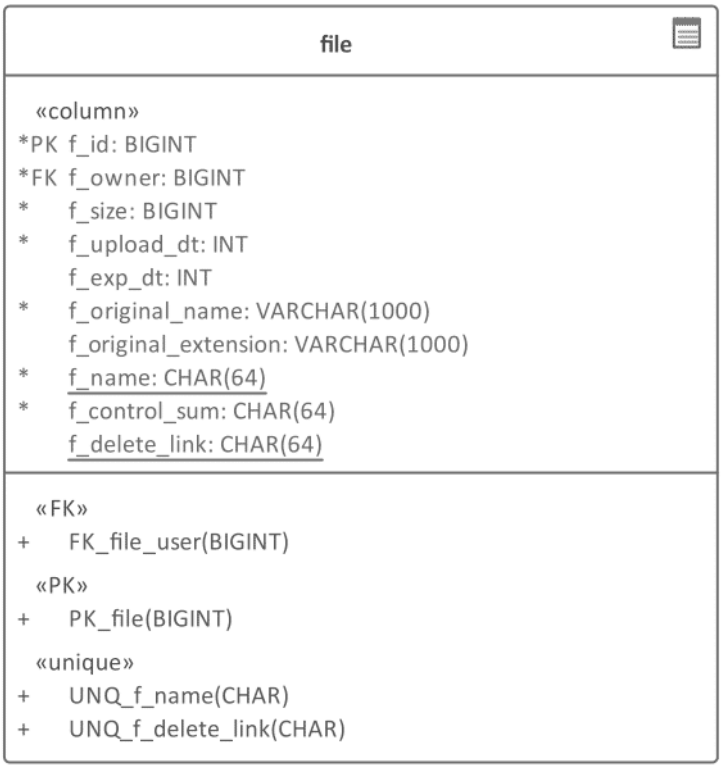


*Рисунок 4.3.е — **Таблица `file` до создания дополнительных индексов***

Предположим, что наиболее часто с данной таблицей будут выполняться такие операции:

- ~~поиск файлов конкретного пользователя (этот индекс создаётся автоматически);~~
- удаление файлов с истёкшим сроком хранения;
- поиск файлов по имени и/или расширению;
- упорядочивание списка файлов по размеру;
- упорядочивание списка файлов по моменту загрузки.

Несмотря на полную самоочевидность результата, всё же проверим на примере удаления файлов с истёкшим сроком хранения, как будет выглядеть план выполнения запроса.

MySQL **Получение плана выполнения запроса**

```
1   EXPLAIN DELETE FROM `file` WHERE `f_exp_dt` <= UNIX_TIMESTAMP()
```

Вполне ожидаемо план показывает, что СУБД придётся сканировать всю таблицу в поиске необходимых записей (значения параметров possible_keys и key равны **NULL**):

select_type	table	type	possible_keys	key	key_len	ref	rows	filtered	Extra
DELETE	file	ALL	NULL	NULL	NULL	NULL	1000000	100.00	Using where

Создадим средствами Sparx Enterprise Architect (SQL-код будет представлен далее) индекс на поле **f_exp_dt**. Теперь план запроса выглядит иначе:

select_type	table	type	possible_keys	key	key_len	ref	rows	filtered	Extra
DELETE	file	ALL	IDX_f_exp_dt	IDX_f_exp_dt	4	NULL	1000000	50.00	Using where

И остаётся провести эксперимент. Предварительно удалим в реально работающей базе данных индекс на поле **f_exp_dt**, при миллионе записей в таблице определим медианное время (на 100 попытках) выполнения запроса по поиску всех файлов с истёкшим сроком хранения. Затем снова создадим индекс на поле **f_exp_dt** и снова определим это время. Абсолютные величины нас не интересуют (т.к. они сильно зависят от аппаратного обеспечения), но конечный результат не может не радовать: скорость выполнения запроса увеличилась в 121 раз.

Без индекса, с	С индексом, с	Разница, раз
7.00951E-03	5.79357E-05	120.98775021274

Полностью аналогичные операции стоит выполнить для полей **f_size** и **f_upoload_dt** (для ускорения упорядочивания списка файлов по размеру и по моменту загрузки соответственно).

В случае с поиском файлов по имени и/или расширению стоит предположить, что поиск по расширению (типу) будет выполняться чаще и, если мы не хотим создавать несколько индексов, выгоднее будет поставить поле **f_original_extension** в индексе на первое место, а поле **f_original_name** — на второе. Ранее{52} было подробно объяснено, почему порядок полей в составных индексах критичен.

После выполнения всех необходимых процедур в Sparx Enterprise Architect таблица **file** принимает следующий вид (см. рисунок 4.3.f).

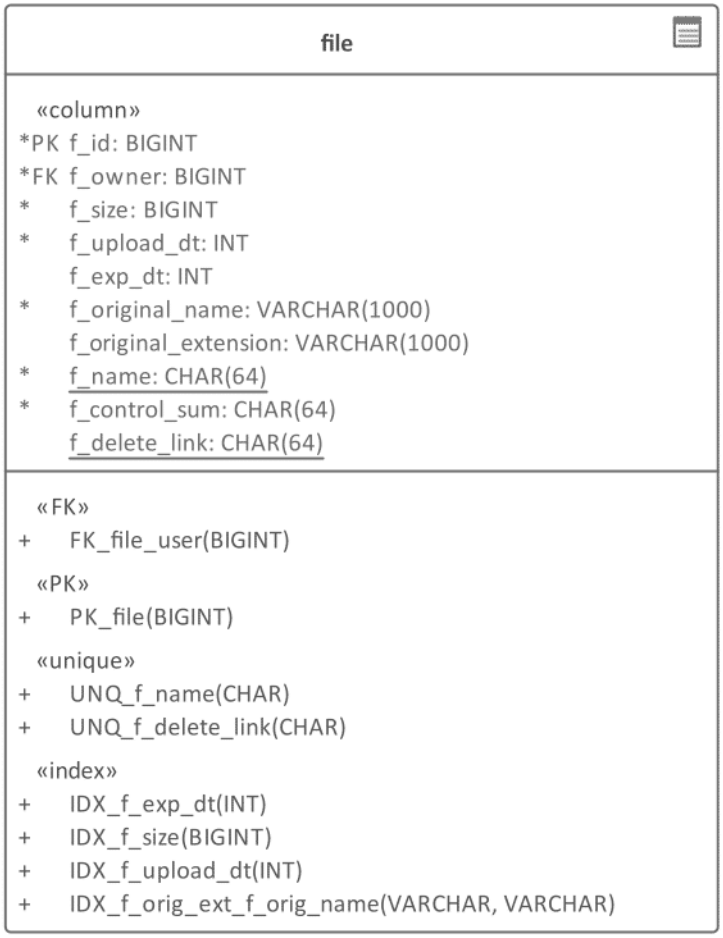


*Рисунок 4.3.f — **Таблица `file` после создания дополнительных индексов***

Если же вы предпочитаете создавать индексы с помощью SQL-кода, это можно сделать следующим образом:

MySQL **SQL-код для создания только что рассмотренных индексов**

```
ALTER TABLE `file`
 ADD INDEX `IDX_f_exp_dt` (`f_exp_dt` ASC);

ALTER TABLE `file`
 ADD INDEX `IDX_f_size` (`f_size` ASC);

ALTER TABLE `file`
 ADD INDEX `IDX_f_upload_dt` (`f_upload_dt` ASC);

ALTER TABLE `file`
 ADD INDEX `IDX_f_orig_ext_f_orig_name`
          (`f_original_extension` ASC,`f_original_name` ASC);
```

Теперь остаётся повторить аналогичные рассуждения и исследования для остальных таблиц базы данных (что и является одним из заданий для самостоятельной проработки, см. задание 4.3.f{330}).

Легко заметить, что с технической точки зрения в создании индексов нет ничего сложного, сложность здесь состоит в необходимости проводить много рассуждений и экспериментов, а также тщательно изучать техническую документацию к конкретной версии конкретной СУБД.

Настройки СУБД

В нашем примере мы используем MySQL, и можно смело сказать, что нам очень повезло — MySQL как единое целое конфигурируется через SQL-запросы, переменные окружения и специальные файлы опций.

В случае использования других СУБД нам пришлось бы конфигурировать несколько отдельных приложений, использовать не только SQL-запросы, переменные окружения и файлы опций (в разных форматах), но и реестр Windows, и даже отдельные внешние сервисы.

Но даже в MySQL настроек очень много[212]. Для наглядности приведём список файлов опций, которые использует эта СУБД.

Файл	Назначение
%WINDIR%\my.ini, %WINDIR%\my.cnf	Глобальные настройки
C:\my.ini, C:\my.cnf	Глобальные настройки
BASEDIR\my.ini, BASEDIR\my.cnf	Глобальные настройки
defaults-extra-file	Используется при запуске с ключом --defaults-extra-file
DATADIR\mysqld-auto.cnf	Постоянные системные переменные для SET PERSIST или SET PERSIST_ONLY

Если выполнить SQL-запрос `SHOW VARIABLES`, вы увидите более 500 различных настроек, которые MySQL использует в своей работе. Иногда для их анализа и лучшего понимания может пригодиться SQL-запрос `SHOW STATUS`, который показывает более 300 параметров текущего состояния MySQL.

Для примера изменения настроек предположим, что вся СУБД находится полностью в нашем распоряжении (т.е. мы не сломаем логику работы других пользователей) и переключим кодировки по умолчанию.

Это можно сделать с помощью SQL-запросов тремя способами:

MySQL SQL-код для временного переключения кодировок по умолчанию

```
-- Способ 1:
SET character_set_server = utf8mb4
SET collation_server = utf8mb4_general_ci

-- Способ 2:
SET NAMES utf8mb4 COLLATE utf8mb4_general_ci;

-- Способ 3 (настройки сохраняются при перезапуске сервера):
SET PERSIST character_set_server = utf8mb4
SET PERSIST collation_server = utf8mb4_general_ci
```

Также можно отредактировать файл my.ini и добавить в группу опций `[mysqld]` такие строки:

```
character_set_server = utf8mb4
collation_server = utf8mb4_general_ci
```

После чего перезапустить сервис MySQL.

Остаётся напомнить универсальный совет: не меняйте никакие настройки СУБД, если вы не понимаете совершенно чётко, что именно, почему, зачем и как вы делаете; т.к. в большинстве случаев настройки СУБД по умолчанию «оптимизированы» под широкий спектр типичных решений, к которым, скорее всего, и относится ваша база данных.

---

212 Для начала стоит ознакомиться с вот этими разделами документации: https://dev.mysql.com/doc/refman/8.0/en/program-options.html и https://dev.mysql.com/doc/refman/8.0/en/server-administration.html.

Задание 4.3.c: доработайте базу данных «Банк»{408} так, чтобы устранить все обнаруженные в ней недостатки проектирования на физическом уровне (насколько это возможно без привлечения «гипотетического заказчика» для получения ответов на возникающие вопросы).

Задание 4.3.d: ранее{321} в данной главе был рассмотрен пример SQL-кода для создания пользователей в ролях «Приложение» и «Администратор» и управления их правами. Напишите аналогичный SQL-код для ролей «Гость», «Пользователь» и «Модератор».

Задание 4.3.e: ранее{323} в данной главе было рассмотрено создание хранимой процедуры, управляющей кодировками всех интересующих нас объектов базы данных. Также было отмечено{324}, что аналогичного эффекта можно было добиться и без хранимой процедуры. Напишите SQL-код, который выполняет те же действия, но не является частью хранимой процедуры.

Задание 4.3.f: ранее{325} в данной главе мы провели серию исследований и экспериментов по работе с индексами на таблице `file`. Повторите аналогичные исследования и эксперименты для остальных таблиц базы данных файлообменного сервиса.

4.4. ОБРАТНОЕ ПРОЕКТИРОВАНИЕ

4.4.1. ЦЕЛИ И ЗАДАЧИ ОБРАТНОГО ПРОЕКТИРОВАНИЯ

Как следует из самого термина — обратное проектирование помогает нам на основе уже существующей и работающей базы данных получить информацию о её структуре, настройках, логике работы и т.д.

Зачем нужна эта информация? Вариантов может быть несколько:

- Заказчик передал нам для поддержки и доработки проект, ранее реализованный другим исполнителем.
- Необходимо внести правки в имеющийся проект, но в силу тех или иных причин документация не даёт нам ответов на имеющиеся вопросы.
- Мы изучаем старый проект для того, чтобы перенести часть его функциональности в новый.
- И иные более редкие случаи.

В процессе прямого проектирования мы всегда вынуждены проходить все три уровня (инфологический, даталогический, физический), причём неоднократно.

При обратном проектировании ситуация выглядит иначе.

Во-первых, нам приходится намного реже «возвращаться» на предыдущий уровень, т.к. при правильной организации работы и использовании подходящих инструментов мы можем за один раз собрать всю необходимую информацию.

Во-вторых, нам достаточно часто не нужно получать модели всех уровней: например, нет никакого смысла строить инфологическую модель уже существующей базы данных, если мы хотим решить пару проблем с производительностью и настроить индексы и/или методы доступа.

Если свести вышесказанное к паре кратких формулировок, получится:

- цель обратного проектирования — получить о существующей базе данных информацию, достаточную для выполнения предстоящей работы;
- задача обратного проектирования — собрать необходимую информацию в полном объёме и представить её в удобной для восприятия и дальнейшего использования форме.

Немного за рамками данной темы (но всё же достойно упоминания) находится анализ базы данных и настроек СУБД при устранении неких технических проблем и оптимизации производительности.

Строго формально такой анализ тоже можно отнести к обратному проектированию баз данных, но на практике его таковым не считают потому, что в процессе такого анализа ставятся достаточно узкие, специфические технические задачи.

И для решения этих задач с одной стороны нужна очень специфическая информация (которой может быть совершенно недостаточно для понимания общей структуры базы данных, её связи с предметной областью и т.д.), а с другой стороны такой информации может понадобиться очень много (намного больше, чем нужно было бы, если бы мы занимались «классическим» обратным проектированием).

В любом случае — если вы исследуете работающую систему (и базу данных как её часть), вы собираете информацию.

Вид, объём, форма представления, способы извлечения этой информации могут быть очень разными, потому остро встаёт вопрос о поиске наиболее подходящих инструментов.

Задание 4.4.а: сформулируйте список вопросов, которые чаще всего возникали у вас при анализе базы данных «Банк»[{408}].

4.4.2. ИНСТРУМЕНТЫ И ТЕХНИКИ ОБРАТНОГО ПРОЕКТИРОВАНИЯ

Поскольку здесь мы говорим именно об *обратном* проектировании, начнём с физического уровня.

Как правило, здесь речь идёт о разнообразных диагностических инструментах, позволяющих нам получить информацию о настройках СУБД, собрать статистику её работы, оценить параметры производительности, увидеть возникающие ошибочные ситуации и определить их первоисточник.

С такими задачами лучше всего справляются узкоспециализированные инструменты, разрабатываемые производителями СУБД или специальные (как правило, весьма недешёвые) коммерческие инструменты. Однако огромное количество данных можно собрать обычными SQL-запросами. В следующей главе будет представлен соответствующий пример.

С даталогическим уровнем ситуация выглядит чуть-чуть полегче потому, что большинство инструментов прямого проектирования баз данных также поддерживает и обратное проектирование: инструмент подключается к СУБД и извлекает оттуда информацию о таблицах, связях, индексах, хранимых процедурах и т.д.

Основная сложность здесь в удобстве представления информации. Эта проблема не настолько остро проявляет себя на физическом уровне, т.к. там мы (как правило) извлекаем небольшой объём данных, которыми будем пользоваться относительно недолго, потому неудобство представления этих данных можно «перетерпеть».

Но если мы поднялись на даталогический уровень, то (скорее всего) нам нужно очень много информации, и мы планируем использовать её достаточно долгое время. Тем более, что часть данных с физического уровня также попадёт в нашу выборку, потому что некоторые структуры базы данных невозможно описать корректно, если проигнорировать низкоуровневые технические детали.

И здесь нас ждёт большое разочарование, т.к. ни узкоспециализированные{305}, ни универсальные{306} инструменты до сих пор не умеют представлять собранную информацию столь же удобно для восприятия человеком, как могла бы выглядеть изначально созданная и оформленная человеком схема базы данных.

Ситуация является более-менее «терпимой», если мы говорим о базе данных с парой десятков таблиц, но когда их число измеряется сотнями, в первый момент вы получите совершенно нечитаемую мешанину из хаотично расположенных таблиц и связей. И придётся потратить немало времени на приведение этой картины в удобную для дальнейшей работы форму.

На рисунке 4.4.а представлен результат обратного проектирования относительно небольшой базы данных (порядка 100 таблиц), выполненного с помощью DBeaver[213]. Да, инструмент сделал всё возможное, чтобы хоть как-то расположить данные в пригодном для восприятия виде. И «пригодно» получилось, но «удобно», к сожалению, нет. Потому здесь в любом случае предстоит много ручной работы.

Отдельного упоминания заслуживает ситуация, в которой после выполнения обратного проектирования и внесения в модель базы данных необходимых правок мы бы хотели автоматически спроецировать эти изменения на реальную работающую базу данных.

Несмотря на то, что многие инструменты поддерживают подобную функциональность, всё зависит от сути изменений. Так, например, нет ничего сложного в том, чтобы добавить в таблицу поле, или удалить поле, или переименовать представление.

213 «DBeaver» [https://dbeaver.io]

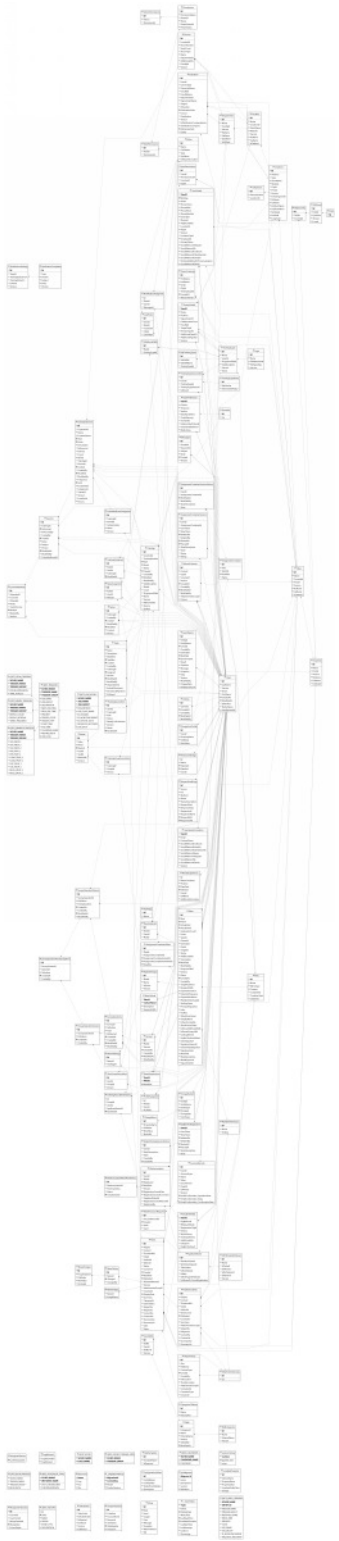


*Рисунок 4.4.а — **Результат обратного проектирования***

Но как, выполнить перемещение поля из одной таблицы в другую (без потери данных)? Как объединить несколько полей в одно или разделить одно на несколько (опять же — без потери данных)? Как правило, ни один инструмент не предоставляет готовых решений потому, что алгоритм выполнения такой операции в каждом отдельном случае будет уникальным и очень специфичным.

Потому здесь снова предстоит выполнить много ручной работы (как правило по написанию соответствующего SQL-кода; возможно, с созданием неких временных промежуточных состояний базы данных).

И, наконец, буквально пара слов об инфологическом уровне. Если мы вынуждены в процессе обратного проектирования подняться так высоко, скорее всего, нам предстоит или некая глобальная переработка существующего проекта, или вовсе разработка нового.

Потому можно смело говорить о том, что здесь мы, фактически, возвращаемся к прямому проектированию, только вместо заказчика информацию нам предоставляет некий инструмент. Полученную информацию мы оформляем в удобном для восприятия виде и начинаем обсуждать с заказчиком — как мы и делали бы при классическом прямом проектировании.

Задание 4.4.b: с использованием готового SQL-кода по созданию базы данных «Банк»{412} выполните такое создание в СУБД MS SQL Server и Oracle, а затем произведите обратное проектирование полученной базы данных с применением любого удобного вам инструмента.

4.4.3. ПРИМЕР ОБРАТНОГО ПРОЕКТИРОВАНИЯ

Поскольку представленные здесь операции будут полностью идентичными для всех объектов базы данных, рассмотрим их на двух показательных примерах: исследовании хранимой процедуры и таблицы.

Как и было сказано в предыдущей главе, много информации можно получить выполнением SQL-запросов в любом инструменте, позволяющем это делать (даже в банальном консольном клиенте, хоть это и худший выбор из возможных). Поскольку мы используем MySQL, выполним эту операцию в MySQL Workbench.

Получить список имеющихся хранимых процедур и список таблиц в базе данных можно такими SQL-запросами:

MySQL **Получение списка имеющихся хранимых процедур и списка таблиц в базе данных**

```
1   -- Получение списка хранимых процедур:
2   SHOW PROCEDURE STATUS WHERE `db` = DATABASE()
3
4   -- Получение списка таблиц:
5   SELECT `table_name`
6     FROM `information_schema`.`tables`
7    WHERE `table_schema` = DATABASE()
8      AND `table_type` = 'BASE TABLE'
```

Когда мы нашли имена интересующих нас хранимой процедуры (**`SET_ENCODING_AND_STORAGE_ENGINE_TO_ALL_TABLES`**) и таблицы (**`file`**), мы можем получить код процедуры и много информации о таблице.

MySQL **Получение кода хранимой процедуры и информации о таблице**

```
1   -- Получение исходного кода создания хранимой процедуры:
2   SHOW CREATE PROCEDURE `SET_ENCODING_AND_STORAGE_ENGINE_TO_ALL_TABLES`
3
4   -- Получение исходного кода создания таблицы:
5   SHOW CREATE TABLE `file`
6
7   -- Получение списка индексов таблицы:
8   SHOW INDEX FROM `file`
```

Исходный код создания хранимой процедуры будет выглядеть в точности так, как мы его писали для её создания — он рассмотрен ранее{325}.

Исходный код создания таблицы будет выглядеть так. Обратите внимание, что все комментарии, которые мы тщательно прописывали в процессе создания базы данных, здесь по-прежнему сохранены. И они могут очень сильно сэкономить время кому-то, кто впервые видит структуру этой таблицы.

MySQL **Исходный код создания таблицы `file`**

```
1   CREATE TABLE `file` (
2     `f_id` bigint(20) unsigned NOT NULL AUTO_INCREMENT
3       COMMENT 'Идентификатор файла.',
4     `f_owner` bigint(20) unsigned NOT NULL
5       COMMENT 'Владелец файла.',
6     `f_size` bigint(20) unsigned NOT NULL
7       COMMENT 'Размер (в байтах).',
8     `f_upload_dt` int(11) NOT NULL
9       COMMENT 'Дата и время добавления (с точностью до секунды).',
```

```
10    `f_exp_dt` int(11) DEFAULT NULL
11      COMMENT 'Срок хранения (до какой даты и какого времени, с точностью
12               до секунды). NULL, если хранить бессрочно.',
13    `f_original_name` varchar(1000) NOT NULL
14      COMMENT 'Исходное имя (без расширения).',
15    `f_original_extension` varchar(1000) DEFAULT NULL
16      COMMENT 'Исходное расширение.',
17    `f_name` char(64) NOT NULL
18      COMMENT 'Имя на сервере (sha256-хэш).',
19    `f_control_sum` char(64) NOT NULL
20      COMMENT 'Контрольная сумма (sha256-хэш).',
21    `f_delete_link` char(64) DEFAULT NULL
22      COMMENT 'Ссылка на удаление (для незарегистрированных пользователей,
23               sha256-хэш).',
24    PRIMARY KEY (`f_id`),
25    UNIQUE KEY `UNQ_f_name` (`f_name`),
26    UNIQUE KEY `UNQ_f_delete_link` (`f_delete_link`),
27    KEY `IDX_f_exp_dt` (`f_exp_dt`),
28    KEY `IDX_f_size` (`f_size`),
29    KEY `IDX_f_upload_dt` (`f_upload_dt`),
30    KEY `IDX_f_orig_ext_f_orig_name` (`f_original_extension`,
31                                      `f_original_name`),
32    KEY `FK_file_user` (`f_owner`),
33    CONSTRAINT `FK_file_user` FOREIGN KEY (`f_owner`)
34    REFERENCES `user` (`u_id`) ON DELETE NO ACTION ON UPDATE NO ACTION
35  ) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='Хранимый на сервере файл.'
```

Что касается индексов, то их «исходный код» (если можно так выразиться) уже представлен в исходном коде создания таблицы. Но выполнение запроса по показу списка индексов может дать нам немного больше информации. Его результат представляет собой вот такую таблицу.

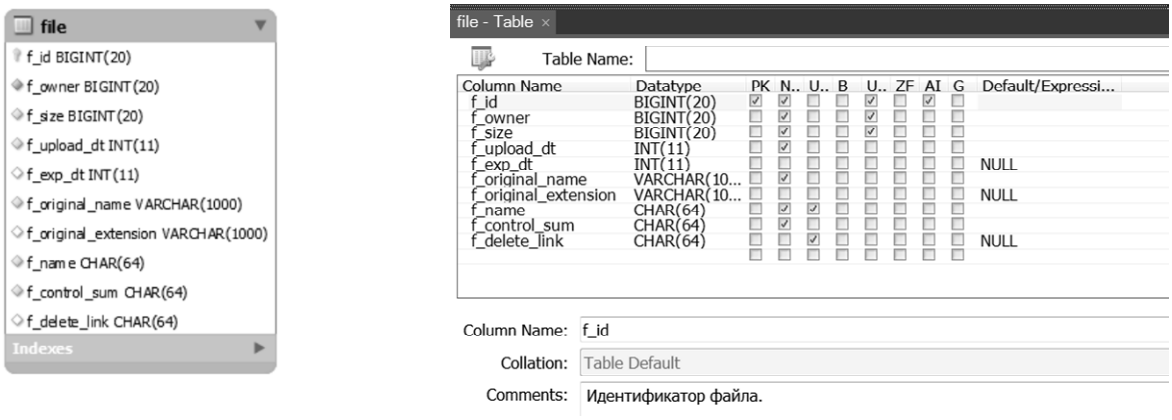
Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type
0	PRIMARY	1	f_id	A	0	NULL	NULL		BTREE
0	UNQ_f_name	1	f_name	A	0	NULL	NULL		BTREE
0	UNQ_f_delete_link	1	f_delete_link	A	0	NULL	NULL	YES	BTREE
1	IDX_f_exp_dt	1	f_exp_dt	A	0	NULL	NULL	YES	BTREE
1	IDX_f_size	1	f_size	A	0	NULL	NULL		BTREE
1	IDX_f_upload_dt	1	f_upload_dt	A	0	NULL	NULL		BTREE
1	IDX_f_orig_ext_f_orig_name	1	f_original_extension	A	0	NULL	NULL	YES	BTREE
1	IDX_f_orig_ext_f_orig_name	2	f_original_name	A	0	NULL	NULL		BTREE
1	FK_file_user	1	f_owner	A	0	NULL	NULL		BTREE

Здесь мы видим информацию и последовательность расположения полей в индексе, кодировку, информацию о проиндексированных записях и многое-многое другое.

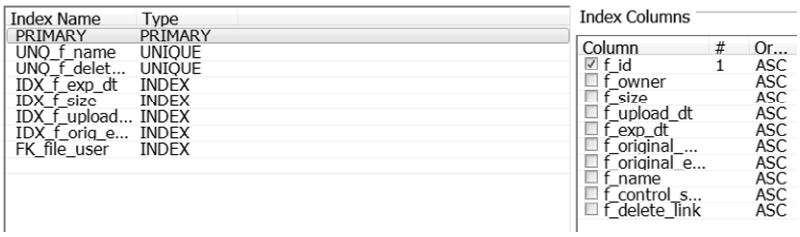
Но всю ту же самую информацию можно получить и выполним импорт всей базы данных целиком средствами MySQL Workbench. В меню выбираем Database → Reverse Engineer и после выполнения показанных инструкций получаем результат, представленный на рисунке 4.4.b.

Таблица `file`

Детальное описание таблицы `file`



Индексы таблицы `file`



Хранимая процедура

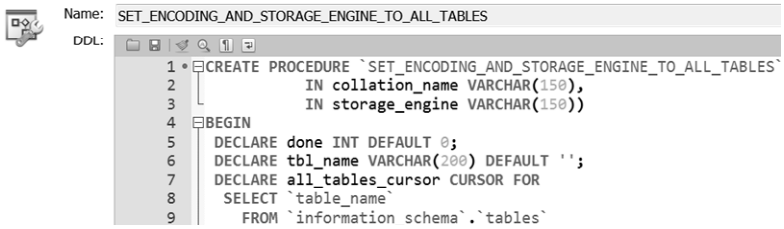


*Рисунок 4.4.b — **Результат импорта в MySQL Workbench***

Фактически, мы уже находимся на стадии обратного проектирования до даталогического уровня. На рисунке 4.4.c представлена обща картина (вся база данных).

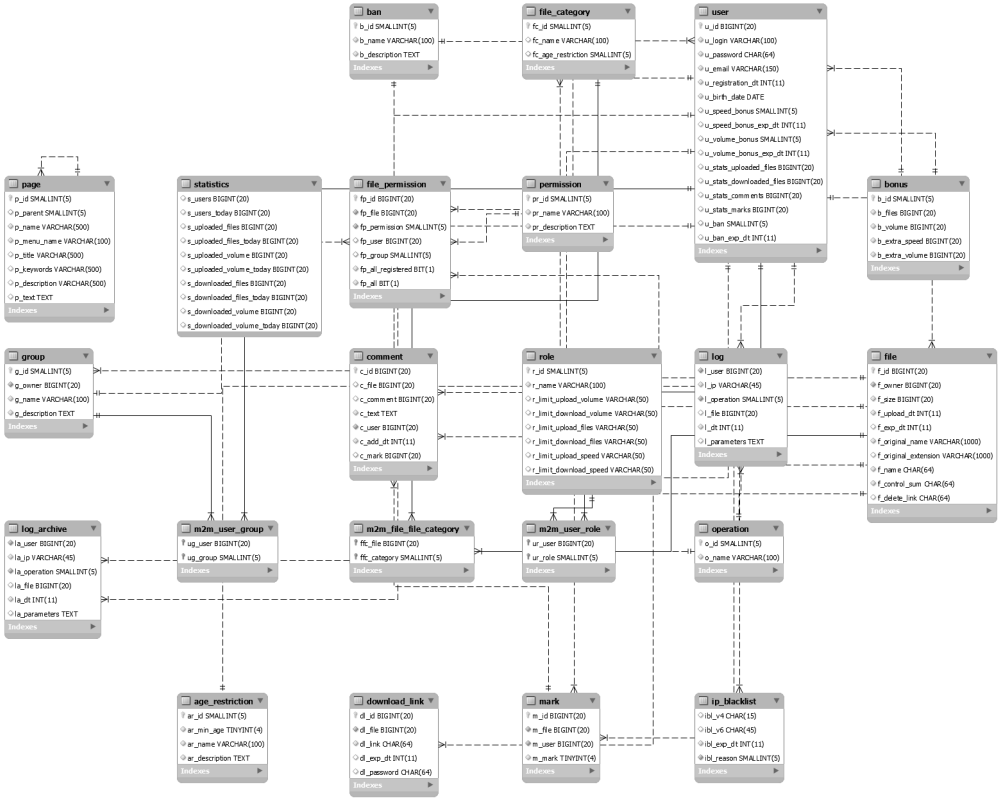


*Рисунок 4.4.c — **Результат импорта всей базы данных в MySQL Workbench***

Выполнив последний шаг (для краткости — только для таблицы `file`), мы получим знакомое нам по началу проектирования словесное описание следующего вида.

Файл:

- Идентификатор файла.
- Владелец файла.
- Размер (в байтах).
- Дата и время добавления (с точностью до секунды).
- Срок хранения (до какой даты и какого времени, с точностью до секунды). NULL, если хранить бессрочно.
- Исходное имя (без расширения).
- Исходное расширение.
- Имя на сервере (sha256-хэш).
- Контрольная сумма (sha256-хэш).
- Ссылка на удаление (для незарегистрированных пользователей, sha256-хэш).

Остаётся получить такие же словесные описания для остальных таблиц — и на этом процесс обратного проектирования будет полностью завершён.

Задание 4.4.c: сравните модели базы данных «Банк»{408} и результат обратного проектирования, полученный при выполнении задания 4.4.b{335}. Составьте список отличий с пометками о том, какой вариант предоставляет больше информации и более удобен для восприятия.

ДОПОЛНИТЕЛЬНЫЕ ОБЪЕКТЫ И ПРОЦЕССЫ БАЗ ДАННЫХ

5.1. ПРЕДСТАВЛЕНИЯ

5.1.1. ОБЩИЕ СВЕДЕНИЯ О ПРЕДСТАВЛЕНИЯХ

До сих пор мы осознанно не использовали представления, чтобы не усложнять рассматриваемые примеры, однако в реальных базах данных представления встречаются повсеместно, и их применение позволяет получить широкий спектр преимуществ.

Представление (view[214]) — виртуальная производная переменная отношения{24}, значением которой является результат вычисления реляционного выражения (выполнения запроса), заданного при создании представления (такое выражение должно ссылаться хотя бы на одну переменную отношения).

Упрощённо: SQL-запрос, который можно выполнять, обращаясь к нему по заранее указанному имени.

Сразу же рассмотрим ещё одно очень важное определение.

Материализованное представление (materialized view[215]) — производная переменная отношения{24}, значением которой является сохранённый результат заранее вычислен-

214 **View** — a derived relvar that's virtual, not real (contrast snapshot). The value of a given view at a given time is the result of evaluating a certain relational expression — the view defining expression, specified when the view itself is defined — at the time in question. The view defining expression must mention at least one relvar, for otherwise the view wouldn't be, specifically, a variable as such. («The New Relational Database Dictionary», C.J. Date)

215 **Materialized view, snapshot** — a derived relvar that's real, not virtual (contrast view). The value of a given snapshot at a given time is the result of evaluating a certain relational expression — the snapshot defining expression, specified when the snapshot itself is defined — at some time prior to the time in question: to be precise, at the most recent «refresh time». The snapshot is «refreshed» (i.e., the snapshot defining expression is reevaluated and the result assigned as the new current value of the snapshot) on explicit user request or, more usually, when some prescribed event occurs, such as the passing of a certain interval of time. Note: The snapshot defining expression must mention at least one relvar, for otherwise the snapshot wouldn't be a variable as such. («The New Relational Database Dictionary», C.J. Date)

ного реляционного выражения (выполнения запроса), заданного при создании материализованного представления (такое выражение должно ссылаться хотя бы на одну переменную отношения). Повторное вычисление и сохранение полученного результата происходит согласно правилам, определённым при создании материализованного представления.

Упрощённо: SQL-запрос, который можно выполнять, обращаясь к нему по заранее указанному имени, и результат выполнения которого сохраняется для дальнейшего использования.

Важно! Для описания этого понятия примерно с одинаковой частотой используют как термин «материализованное представление» (materialized view), так и термин «постоянное представление» (persistent view).

Итак, мы выяснили, что представления можно условно поделить на два типа:
- «обычные» представления, результат вычисления которых нигде не сохраняется;
- материализованные представления, результат вычисления которых сохраняется и может быть повторно использован.

Схематично эта разница представлена на рисунке 5.1.а.

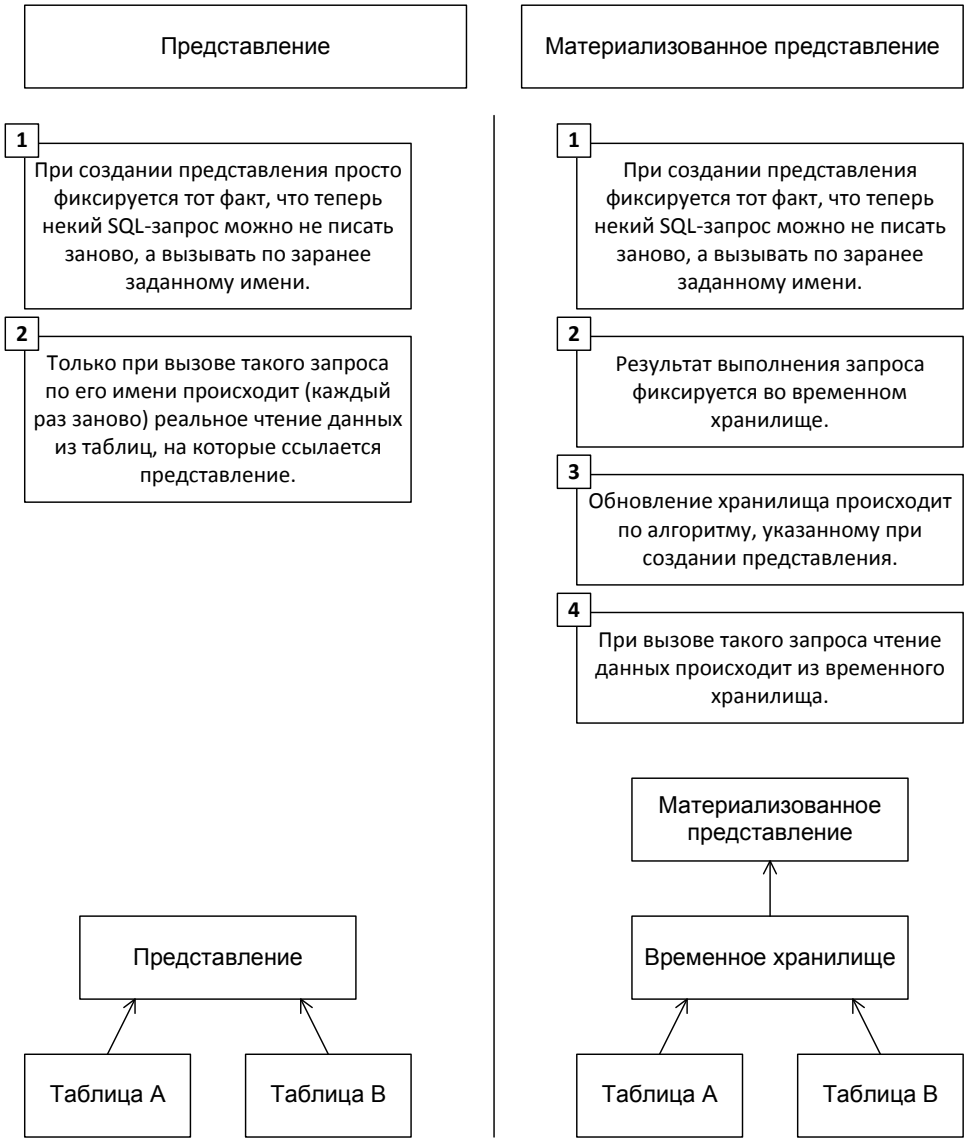


*Рисунок 5.1.а — **Сравнение представлений и материализованных представлений***

Поскольку представления строятся на основе SQL-запросов, следует уточнить, что:

- «основой» для построения представления может быть только **`SELECT`**-запрос (т.е. нельзя построить представление на основе запросов с **`INSERT`**, **`UPDATE`**, **`DELETE`**);
- в некоторых случаях представление может допускать «двунаправленность», т.е. с его использованием можно будет не только читать данные, но и модифицировать их.

Т.н. «двунаправленные» (или «обновляемые») представления до сих пор вызывают много споров. Они реализованы в большинстве СУБД, но их возможности сильно ограничены, критически зависят от определяющего отношение SQL-запроса, и в общем случае использование таких представлений может приводить к неопределённому поведению СУБД.

Гораздо важнее помнить о том, что если с использованием представлений реализуется модель безопасности, необходимо прилагать дополнительные усилия, чтобы гарантированно сделать такие представления **не** «двунаправленными», т.е. заблокировать возможность изменения данных с их использованием.

Казалось бы, если представление — это (упрощённо) всего лишь именованный SQL-запрос, то какой в нём вообще смысл? На самом деле, преимуществ много:

- Упрощение выполнения запросов. Представление может быть построено на SQL-запросе любой сложности. И теперь вместо работы с очень сложным запросом, который может занимать десятки и сотни строк, нам достаточно написать что-то наподобие **`SELECT * FROM представление`**, чтобы получить тот же результат.
- Возможность построения простого и надёжного API[216]. На стадии проектирования базы данных можно предусмотреть набор удобных для использования представлений, которые будут «скрывать» от разработчиков приложений, использующих базу данных, сложные запросы и прочую нетривиальную логику, что значительно упрощает и ускоряет разработку таких приложений, а также снижает количество допущенных при их разработке ошибок.
- Упрощение бизнес-логики. Во многом этот пункт следует из предыдущего, но даже в случае, если мы не создаём полноценный API216, мы можем сформировать набор представлений для наиболее сложных и востребованных случаев (например, в некоем интернет-магазине нужно показывать популярные товары, и алгоритм определения популярных товаров очень сложный и нетривиальный; логично было бы создать представление **`popular_items`**, которое выдавало бы готовый список таких товаров).
- Минимум накладных расходов. Представление («обычное», не материализованное) практически не занимает места в базе данных, потому даже создание сотен и тысяч представлений не приводит к сколь бы то ни было ощутимому увеличению размера базы данных.
- Производительность. В случае использования материализованных представлений мы получаем возможность многократно использовать результаты выполнения SQL-запроса, и выигрыш будет тем более ощутимым, чем более «долгий» запрос мы превратили в материализованное представление, и чем реже мы вынуждены обновлять результаты его выполнения.
- Безопасность. Представления являются объектами базы данных, потому к ним применимы все механизмы СУБД по контролю прав доступа. Поскольку представления дают опосредованный доступ к таблицам базы данных, появляется возможность запретить такой доступ напрямую к таблицам, а разрешить его только через специально созданные и продуманные представления. Дополнительный уровень безопасности можно обеспечить за счёт логики запросов, на которых строятся представления.

---

216 **API** (application programming interface) — a computing interface which defines interactions between multiple software intermediaries. («Wikipedia») [https://en.wikipedia.org/wiki/Application_programming_interface]

Но если всё так прекрасно, почему тогда в каждой базе данных не создать огромное количество представлений? Тому есть несколько причин.

- Избыточность. Иногда представления просто не нужны. Например, база данных небольшая, бизнес-логика простая, модель безопасности тривиальная — здесь представления будут просто лишним слоем абстракции между приложением и базой данных.
- Дополнительный код. Представления не появляются и не существуют сами по себе. Их нужно продумать, создать, протестировать, корректировать при изменении базы данных или бизнес-логики. Это требует дополнительных трудозатрат и повышает вероятность возникновения ошибок.
- Ограниченность возможностей. Использование представлений для модификации данных пусть и реализовано во многих СУБД, но имеет широкий спектр ограничений, потому можно утверждать, что не любую операцию с базой данных можно выполнить с помощью представлений.

Итог очевиден: как и любой другой инструмент, представления могут оказаться очень полезными при их правильном применении, но бездумное их создание и использование столь же вероятно принесёт лишь дополнительные проблемы.

Задание 5.1.а: составьте список представлений, которые стоило бы добавить в базу данных «Банк»[{408}]. Отметьте для каждого представления, какие задачи можно будет решить с его помощью.

5.1.2. СОЗДАНИЕ И ИСПОЛЬЗОВАНИЕ ПРЕДСТАВЛЕНИЙ

Подавляющее большинство инструментов проектирования баз данных позволяет создавать представления как элементы модели базы данных (что повышает удобство процесса проектирования и позволяет автоматизировать генерацию кода). Но поскольку представление, фактически, является «именованным SQL-запросом», всё равно основную работу (т.е. написание запроса) приходится выполнять вручную.

В качестве первого примера рассмотрим следующую ситуацию: допустим, в нашем файлообменном сервисе почему-то появилась необходимость быстро и часто получать список файлов, загруженных в первые выходные дни каждого месяца (первые в каждом месяце субботу и воскресенье).

Вот так будет выглядеть SQL-запрос для решения данной задачи.

MySQL **Получение списка файлов, загруженных в первые выходные каждого месяца**

```
SELECT *
FROM
        (
                SELECT *,
                CASE
                 WHEN WEEKDAY(FROM_UNIXTIME(`f_upload_dt`) —
                 INTERVAL DAY(FROM_UNIXTIME(`f_upload_dt`))-1 DAY) <=
                      WEEKDAY(FROM_UNIXTIME(`f_upload_dt`))
                 THEN WEEK(FROM_UNIXTIME(`f_upload_dt`), 5) -
                      WEEK(FROM_UNIXTIME(`f_upload_dt`) —
                 INTERVAL DAY(FROM_UNIXTIME(`f_upload_dt`))-1 DAY, 5) + 1
                 ELSE WEEK(FROM_UNIXTIME(`f_upload_dt`), 5) -
                      WEEK(FROM_UNIXTIME(`f_upload_dt`) —
                INTERVAL DAY(FROM_UNIXTIME(`f_upload_dt`))-1 DAY, 5)
                END AS `W`,
                WEEKDAY(FROM_UNIXTIME(`f_upload_dt`)) + 1 AS `D`
        FROM `file`
        ) AS `prepared_data`
WHERE (`W` = 1)
  AND ((`D` = 6) OR (`D` = 7))
```

Этот запрос выглядит достаточно громоздко, он сложен для написания и понимания, а его доработка (при условии, что он используется в нескольких местах) с высокой вероятностью приведёт к возникновению ошибок.

Чтобы избежать всех этих неудобств, можно «обернуть» этот запрос в представление, что выглядит следующим образом:

MySQL **Создание представления для получения списка файлов, загруженных в первые выходные каждого месяца**

```
CREATE VIEW `files_on_first_days_off` AS
SELECT *
FROM   ( SELECT *,
                CASE
                 WHEN WEEKDAY(FROM_UNIXTIME(`f_upload_dt`) -
                 INTERVAL DAY(FROM_UNIXTIME(`f_upload_dt`))-1 DAY) <=
                      WEEKDAY(FROM_UNIXTIME(`f_upload_dt`))
                 THEN WEEK(FROM_UNIXTIME(`f_upload_dt`), 5) -
                      WEEK(FROM_UNIXTIME(`f_upload_dt`) -
                 INTERVAL DAY(FROM_UNIXTIME(`f_upload_dt`))-1 DAY, 5) + 1
                 ELSE WEEK(FROM_UNIXTIME(`f_upload_dt`), 5) -
                      WEEK(FROM_UNIXTIME(`f_upload_dt`) -
               INTERVAL DAY(FROM_UNIXTIME(`f_upload_dt`))-1 DAY, 5)
               END AS `W`,
               WEEKDAY(FROM_UNIXTIME(`f_upload_dt`)) + 1 AS `D`
        FROM `file`
        ) AS `prepared_data`
WHERE (`W` = 1)
  AND ((`D` = 6) OR (`D` = 7))
```

И теперь для получения тех же самых данных достаточно выполнить вот такой примитивный запрос:

MySQL **Получение списка файлов, загруженных в первые выходные каждого месяца, через представление**

```
SELECT * FROM `files_on_first_days_off`
```

Если предположить, что такой список файлов нужен для отчётности и может содержать немного устаревшую информацию (допустим, с отставанием в сутки), логично было бы создать материализованное представление с автоматическим обновлением раз в сутки.

К сожалению, MySQL (пока?) не поддерживает материализованные представления, потому в данной СУБД эта задача может быть решена только с использованием т.н. «кэширующей таблицы» и хранимой процедуры{363}, обновляющей эту таблицу (см. задание 5.1.c{346}).

Поддержка материализованных представлений реализована в различных СУБД очень по-разному. Так, например:

- в MS SQL Server нельзя управлять логикой их обновления (оно происходит автоматически при изменении данных);
- в Oracle можно очень гибко управлять логикой их обновления (эта СУБД поддерживает достаточно сложные синтаксис для решения этой задачи);
- в PostgreSQL есть отдельная команда для их обновления.

В любом случае необходимо ориентироваться на конкретную версию конкретной СУБД при выборе оптимального решения.

В качестве второго примера рассмотрим использование представления для управления безопасностью.

Предположим, что понадобилось создать отдельную роль и ограничить возможности находящихся в этой роли пользователей по управлению файлами — им разрешено работать только с файлами с расширениями .jpg, .jpeg, .png, .gif.

Тогда для такой роли необходимо запретить прямой доступ к таблице **`file`**, а вместо этого открыть доступ к специальному представлению:

MySQL **Представление для обеспечения ограниченного доступа к таблице `file`**

```
CREATE VIEW `files_with_jpg_jpeg_png_gif_extensions` AS
SELECT *
  FROM `file`
 WHERE LOWER(`f_original_extension`) IN ('jpg', 'jpeg', 'png', 'gif')
WITH CHECK OPTION
```

Данное представление является «двунаправленным» (обновляемым), т.е. с его помощью можно не только извлекать данные, но и добавлять, и обновлять, и удалять.

Выражение **`WITH CHECK OPTION`** в последней строке запроса предписывает СУБД запрещать операции вставки и обновления, не удовлетворяющие условию **`WHERE`**, т.е. попытка добавить файл с запрещённым расширением или изменить у имеющегося файла расширение на запрещённое завершится ошибкой.

Если мы хотим запретить операции обновления данных, представление можно переписать таким образом:

MySQL **Представление для обеспечения ограниченного доступа к таблице `file`**

```
1   CREATE VIEW `ro_files_with_jpg_jpeg_png_gif_extensions` AS
2   SELECT `f_id`,
3          `f_owner`,
4          `f_size` + 0,
5          `f_upload_dt`,
6          `f_exp_dt`,
7          `f_original_name`,
8          `f_original_extension`,
9          `f_name`,
10         `f_control_sum`,
11         `f_delete_link`
12    FROM `file`
13   WHERE LOWER(`f_original_extension`) IN ('jpg', 'jpeg', 'png', 'gif')
```

Обратите внимание на 4-ю строку запроса (выражение **`f_size` + 0**): очевидно, что при выборке данных добавление 0 к числу не изменяет это число, т.е. извлекаться данные будут корректно, но наличие такого выражения в запросе, на котором построено представление, не позволяет MySQL использовать представление для модификации данных, т.е. поставленная цель достигнута.

Также заметьте, что в данном случае нет необходимости добавлять выражение **`WITH CHECK OPTION`**, т.к. данное представление в принципе не позволяет выполнять никаких операций модификации данных.

Следуя этой же логике, можно сформировать любой набор представлений, ограничивающих доступ к отдельным столбцам таблицы, к записям с определёнными значениями или комбинациями значений и т.д.

Общий алгоритм очень прост: необходимо написать запрос на выборку данных, после чего перед ним добавить конструкцию **`CREATE VIEW `имяпредставления` AS`** — и всё, мы получаем желаемый результат.

Множество дополнительных практических примеров приведено в разделе 3 «Использование представлений» книги[217] «Работа с MySQL, MS SQL Server и Oracle в примерах».

Задание 5.1.b: создайте представления, список которых вы составили при выполнении задания 5.1.a{343}.

Задание 5.1.c: ранее{345} в данной главе было отмечено, что MySQL не поддерживает материализованные представления, потому в данной СУБД задачу по актуализации списка файлов, загруженных в первые выходные дни каждого месяца, можно решить только с использованием т.н. «кэширующей таблицы» и хранимой процедуры, обновляющей эту таблицу. Создайте соответствующие таблицу и хранимую процедуру{363} для её обновления.

---

217 «Работа с MySQL, MS SQL Server и Oracle в примерах» (С.С. Куликов) [http://svyatoslav.biz/database_book/]

5.2. ПРОВЕРКИ

5.2.1. ОБЩИЕ СВЕДЕНИЯ О ПРОВЕРКАХ

С самым простым видом проверок мы уже знакомы — это ограничение **`NOT NULL`**, которое указывается в определении поля таблицы и запрещает вставку в это поле **`NULL`**-значений.

Однако возможности проверок значительно шире, и их использование позволяет очень гибко управлять перечнем допустимых значений полей таблиц.

Проверка (check[218]) — правило, ограничивающее все значения некоторого поля таблицы определёнными условиями.

Упрощённо: условие, которому должно соответствовать значение поля таблицы.

В общем случае проверки могут относиться к таблице целиком (оперируют значениями нескольких полей) или к её отдельным полям (оперируют значением одного поля).

Использование проверок является более простым и лёгким способом контроля консистентности базы данных{72}, чем использование триггеров{350}, т.к. проверки (как правило) не только более просты в описании, но и выполняются быстрее, чем триггеры.

В любой предметной области можно легко определить десятки условий, контроль которых удобно возложить на проверки, например:

- логин и пароль не должны совпадать;
- дата трудоустройства должна быть больше даты рождения как минимум на 18 лет;
- сумма скидки не должна превышать N% от суммы заказа;
- номер паспорта должен иметь указанный формат;
- максимальное время хранения файла на сервере не должно превышать N дней;
- и т.д.

Технические возможности проверок различаются в зависимости от СУБД (от тривиальных условий вида «больше / меньше / (не) равно» до использования регулярных выражений и хранимых функций{363}.

Также рекомендуется помнить о производительности: вычислительно сложные проверки, фактически, оказываются ничем не лучше триггеров по влиянию на скорость операций вставки и обновления данных.

И к ещё одному (пусть и небольшому) недостатку проверок можно отнести тот факт, что в случае нарушения условия проверки мы крайне ограничены в возможностях по формированию информативного сообщения об ошибке. В то время как при использовании триггеров мы можем не только формировать такие сообщения, но и (в некоторых СУБД) порождать исключения и иным образом контролировать весь дальнейший процесс выполнения запроса (вплоть до автоматической корректировки ошибочных данных).

Задание 5.2.а: составьте список проверок, которые стоило бы добавить в базу данных «Банк»{408}. Отметьте для каждой представления, какие задачи из предметной области решаются с их помощью.

---

218 **Check** — a rule that specifies the values allowed in one or more columns of every row of a table. («The New Relational Database Dictionary», C.J. Date)

5.2.2. СОЗДАНИЕ И ИСПОЛЬЗОВАНИЕ ПРОВЕРОК

Использование проверок происходит автоматически — СУБД самостоятельно контролирует описанные проверкой условия и запрещает операцию вставки или обновления данных при их нарушении.

Что же касается создания проверок, то рассмотрим его на двух примерах. Предположим, что заказчик нашего файлообменного сервиса обозначил два следующих бизнес-правила:

- логин и пароль пользователя не должны совпадать;
- максимальное время хранения файла на сервере не должно превышать 500 дней.

Большинство средств проектирования баз данных позволяет добавлять проверки в модель точно так же, как и индексы (в т.ч. ограничения уникальности, получаемые с использованием уникальных индексов{109}). Но даже если выбранное вами средство проектирования не обладает такими возможностями, любая СУБД позволяет добавить в таблицу проверки посредством изменения таблицы (выражение `ALTER TABLE`).

Итак, переходим к нашим примерам.

MySQL поддерживает проверки, начиная с версии 8.0.16. Если вы используете более старую версию, все проверки будут успешно создаваться, но не будут работать (т.е. не будут применяться к поступающим в таблицу данным — словно этих проверок вовсе нет).

В первом случае нам нужно сравнить значения полей `u_login` и `u_password` таблицы `user` и убедиться в том, что они различаются. Ситуация осложняется тем, что пароль уже приходит в виде sha256-хэша, а потому нам нужно вычислить такой хэш от имени пользователя и сравнить полученное значение со значением пароля.

Технически это выглядит следующим образом:

MySQL **Проверка, запрещающая совпадение значений полей `u_login` и `u_password`**

```
ALTER TABLE `user`
 ADD CONSTRAINT `CHK_login_password_mb_different`
  CHECK (SHA2(`u_login`, 256) != `u_password`)
```

Теперь при попытке добавить в базу данных пользователя с одинаковыми логином и паролем мы будем получать сообщение об ошибке:

MySQL **Проверка, запрещающая совпадение значений полей `u_login` и `u_password`**

```
Error Code: 3819.
Check constraint 'CHK_login_password_mb_different' is violated
```

Во втором случае нам нужно удостовериться, что значение поля `f_exp_dt` таблицы `file` находится в пределах 500 дней от значения поля `f_upload_dt`. Здесь мы не можем использовать значение текущей даты, т.к. MySQL запрещает использовать в проверках т.н. «недетерминированные» функции (вызов которых с одними и теми же параметрами может дать разные результат), а функция `CURRENT_DATE()`, возвращающая текущую дату, объективно является недетерминированной.

Технически решение выглядит следующим образом:

MySQL **Проверка, запрещающая хранить файлы более 500 дней**

```
1   ALTER TABLE `file`
2    ADD CONSTRAINT `CHK_max_500_days_store`
3     CHECK (DATEDIFF(FROM_UNIXTIME(`f_exp_dt`),
4                     FROM_UNIXTIME(`f_upload_dt`)) <= 500)
```

Теперь при попытке добавить в базу данных файл, время хранения которого превышает 500 дней, мы будем получать сообщение об ошибке:

MySQL **Сообщение об ошибке**

```
1   Error Code: 3819.
2   Check constraint 'CHK_max_500_days_store' is violated
```

Поскольку при нарушении условия проверки мы не можем сформировать собственное сообщение об ошибке, особенно важно давать проверкам адекватные осмысленные имена. Это позволит намного быстрее диагностировать проблему при разработке и отладке приложений, работающих с вашей базой данных.

Множество более сложных практических примеров приведено в разделе 4 «Использование триггеров» книги[219] «Работа с MySQL, MS SQL Server и Oracle в примерах». Да, этот раздел посвящён именно триггерам, а не проверкам, т.к. триггеры позволяют решать значительно более сложные задачи.

Задание 5.2.b: создайте проверки, список которых вы составили при выполнении задания 5.2.a{347}.

219 «Работа с MySQL, MS SQL Server и Oracle в примерах» (С.С. Куликов) [http://svyatoslav.biz/database_book/]

5.3. ТРИГГЕРЫ

5.3.1. ОБЩИЕ СВЕДЕНИЯ О ТРИГГЕРАХ

Начнём с определения.

Триггер (trigger[220]) — специальный объект базы данных, описывающий перечень действий, которые необходимо автоматически выполнить при наступлении указанного события.

Упрощённо: описание действия, которое нужно автоматически выполнить при определённых условиях.

Триггеры очень по-разному реализованы в различных СУБД, и речь идёт не только о синтаксисе, но и том, с какими событиями можно ассоциировать триггеры, и как именно они выполняются (как обрабатывают данные, связанные с вызвавшим триггер событием).

Начнём с событий. Наиболее классическим вариантом использования триггеров является обеспечение реакции на модификацию данных, т.е. на операции вставки, обновления и удаления.

Очевидно, что при наступлении некоторого события, СУБД может выполнить триггер до, после или вместо соответствующей операции. Для наглядности сведём в одну таблицу все типы триггеров для нескольких СУБД.

		СУБД		
Действие	**До, после, вместо**	**MySQL**	**MS SQL Server**	**Oracle**
Вставка данных	До	+	-	+
	После	+	+	+
	Вместо	-	+	+
Обновление данных	До	+	-	+
	После	+	+	+
	Вместо	-	+	+
Удаление данных	До	+	-	+
	После	+	+	+
	Вместо	-	+	+
Создание таблицы	До	-	-	См. документацию по СУБД
	После	-	+	
	Вместо	-	-	
Удаление таблицы	До	-	-	См. документацию по СУБД
	После	-	+	
	Вместо	-	-	
Авторизация пользователя	До	-	-	См. документацию по СУБД
	После	-	+	
	Вместо	-	-	
Иные операции со структурами базы данных	До	-	-	См. документацию по СУБД
	После	-	-	
	Вместо	-	-	
Иные ситуации	До	-	-	См. документацию по СУБД
	После	-	-	
	Вместо	-	-	

220 **Trigger**, triggered procedure — an action (the «triggered action») to be performed if a specified event (the «triggering event») occurs. A triggered procedure can be thought of as a stored procedure, except that stored procedures must be explicitly invoked, whereas a triggered procedure is invoked automatically whenever the triggering event occurs. («The New Relational Database Dictionary», C.J. Date)

Больше всего нас будут интересовать триггеры, ассоциированные с операциями модификации данных, потому логику их работы представим графически (см. рисунок 5.3.а).

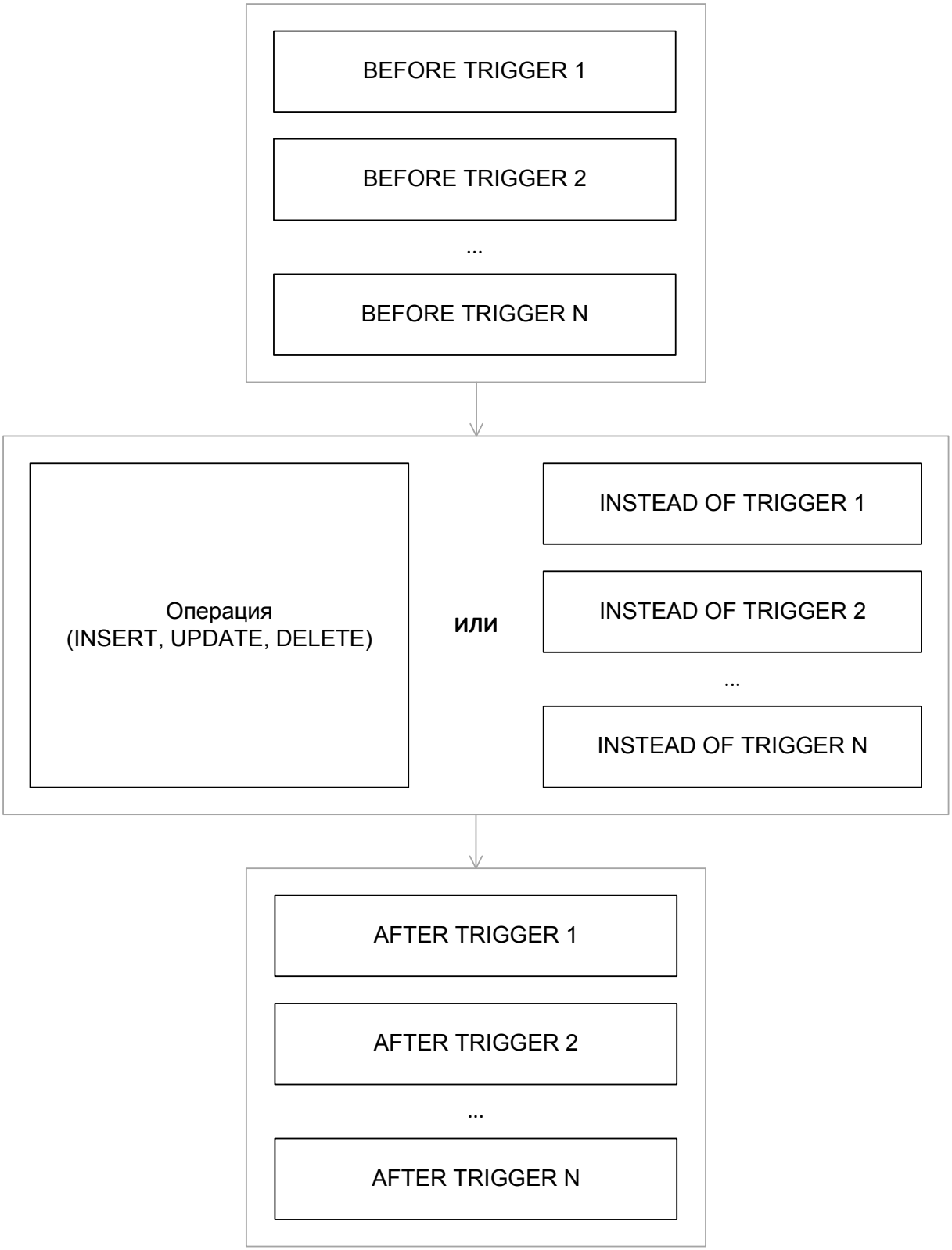


*Рисунок 5.3.а — **Логика работы триггеров***

Отдельно стоит подчеркнуть, что триггеры, выполняемые вместо операции (т.н. **INSTEAD OF** триггеры) на самом деле выполняются *вместо* соответствующей операции, а потому саму операцию необходимо реализовывать внутри триггера и выполнять в том случае, если пройдены все проверки и соблюдены все необходимые условия. Это особенно актуально для MS SQL Server, в котором отсутствуют **BEFORE** триггеры, и их логику приходится реализовывать в **INSTEAD OF** триггерах.

Теперь рассмотрим, как именно триггеры выполняются и обрабатывают данные. В этом контексте триггеры делятся на две группы:

- триггеры уровня строки (row-level triggers) — запускаются каждый раз заново для каждой отдельной строки (записи таблицы), затронутой SQL-запросом;
- триггеры уровня запроса (statement-level triggers) — запускаются один раз для всего SQL-запроса.

Поясним это графически (см. рисунок 5.3.b).

Представим, что в некоторой таблице нам нужно увеличить цену товара на 20% для всех товаров дешевле 50 (валюта не важна, потому просто оставим значение). И допустим, что в таблице есть три записи, удовлетворяющих условию запроса.

Сам запрос может выглядеть так:

MySQL **Увеличение на 20% цены товаров, цена который сейчас меньше 50**

```
1   UPDATE `item`
2      SET `i_price` = `i_price` + (`i_price` * 0.2)
3    WHERE `i_price` < 50
```

Триггер уровня троки будет активирован три раза (отдельно для каждой строки). С помощью специальных ключевых слов **OLD** и **NEW** в большинстве СУБД триггеру будут доступны старые и новые значения полей строки, для которой он вызван.

Триггер уровня запроса будет активирован один раз, и ему будет доступна информация обо всех затронутых обновлением строках. Поскольку строк много, здесь уже невозможно использовать подход с ключевыми словами OLD и NEW (они позволяют работать только с одной строкой), но (на примере логики MS SQL Server) можно использовать специальные «виртуальные» таблицы **DELETED** (содержит данные до обновления) и **INSERTED** (содержит данные после обновления).

Порядок строк в таблицах **DELETED** и **INSERTED** может не совпадать (как специально показано на рисунке 5.3.b), потому в случае изменения значения первичного ключа мы должны полагаться на значения других уникальных полей (если они есть), чтобы сопоставить строки в их старом и новом состоянии. Если в таблице нет иных уникальных полей, но нам необходимо сопоставить строки в их старом и новом состоянии, то для СУБД, не поддерживающих триггеры уровня ряда, эта задача не имеет решения.

Из дополнительных особенностей реализации триггеров можно отметить следующее:

- некоторые СУБД позволяют создать не более одного триггера «определённого типа» на таблицу (например, не более одного **BEFORE INSERT** триггера), другие же СУБД позволяют создавать много «однотипных» триггеров и даже контролировать последовательность их выполнения;
- в различных СУБД «пространство имён» триггеров может быть ограничено таблицей (т.е. для двух разных таблиц можно создать одноимённые триггеры) или всей базой данных (тогда имена всех триггеров во всей базе данных должны различаться);
- возможности и логика работы «экзотических» триггеров (не связанных с операциями модификации данных) совершенно уникальны для каждой СУБД и требуют тщательного изучения документации.

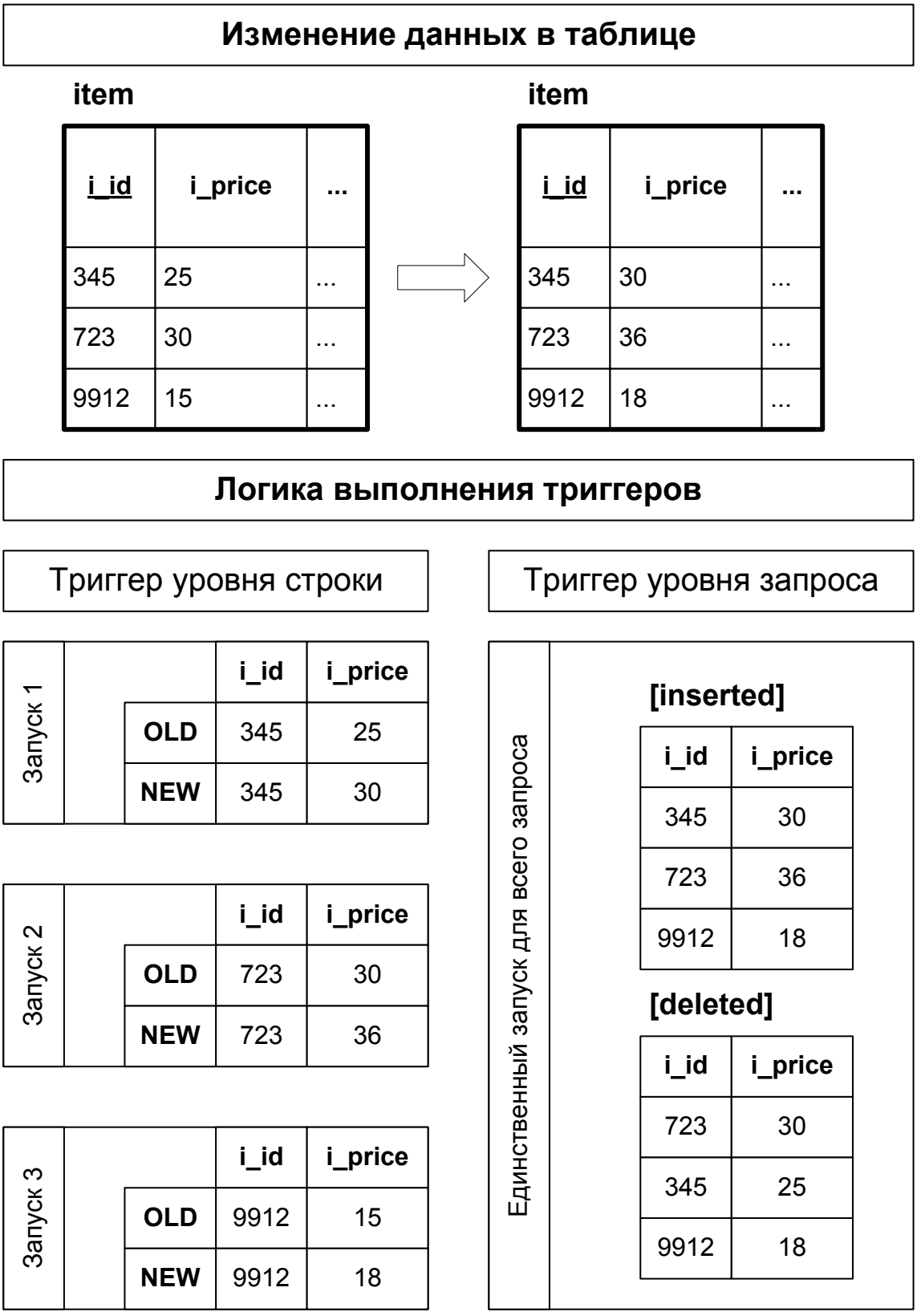


*Рисунок 5.3.b — **Различие работы триггеров уровня строки и уровня запроса***

С помощью триггеров в общем случае решается следующий спектр задач:

- организация каскадных операций{73} (если используемый метод доступа{33} таковые не поддерживает) или реализация более сложной логики каскадных операций, чем предоставляет СУБД;
- обновление данных кэширующих (или агрегирующих) полей и таблиц (пример будет рассмотрен далее, это — один из самых частых случаев использования триггеров);
- обеспечение консистентности{72}, т.е. контроль и изменение таких значений полей, которые находятся в строгой зависимости от значений других полей (или иных условий);
- контроль операций модификации данных для обеспечения выполнения правил бизнес-логики (например, запрет удаления последнего пользователя в роли «администратор» или запрет удаления такой роли у такого пользователя, если бизнес-логика предписывает, что «в системе всегда должен быть хотя бы один администратор»);
- контроль мощности связей{57} в случае, когда «стандартных» вариантов (один к одному, один ко многим, многие ко многим) недостаточно, и необходимо обеспечить более сложное поведение;
- контроль формата и значений данных в случае, если СУБД не поддерживает проверки{347}, или необходимо сформировать информативные сообщения об ошибках, или логика проверки выходит за рамки того, что предоставляют стандартные средства СУБД;
- прозрачное исправление ошибок в данных (в тех редких случаях, когда это возможно; например — приведение ФИО к формату «Фамилия Имя Отчество», т.е. все слова написаны в нижнем регистре с заглавное буквы, даже если изначально данные пришли, например, в виде «ФАМИЛИЯ имя оТчЕсТвО»).

К недостаткам триггеров стоит отнести их влияние на производительность. И оно тем сильнее, чем более сложные операции выполняются внутри триггеров (а такие операции могут включать обращение к другим таблицам, сложные вычисления и т.д.)

И тем не менее триггеры очень распространены. С их помощью решается широкий спектр прикладных задач (особенно в плане обеспечения защищённости данных{17}).

Задание 5.3.а: составьте список триггеров, которые стоило бы добавить в базу данных «Банк»{408}. Отметьте для каждого триггера, какие задачи решаются с его помощью.

5.3.2. СОЗДАНИЕ И ИСПОЛЬЗОВАНИЕ ТРИГГЕРОВ

Как и в случае с проверками{347}, использование триггеров происходит автоматически — СУБД самостоятельно запускает их в соответствующие моменты времени (при наступлении указанных событий), нам лишь нужно написать код, реализующий требуемые действия.

Большинство средств проектирования баз данных позволяет добавить в модель соответствующий объект, но поскольку триггер определяется своим SQL-кодом, этот код всё равно придётся писать вручную.

В качестве примера рассмотрим решение двух задач (обе они были упомянуты в предыдущей главе):

- обновление данных агрегирующей таблицы;
- обеспечение выполнения правила бизнес-логики «в системе всегда должен быть хотя бы один администратор»).

Решение первой задачи сводится к обновлению данных в агрегирующей таблице `statistics` (см. рисунок 5.3.c).

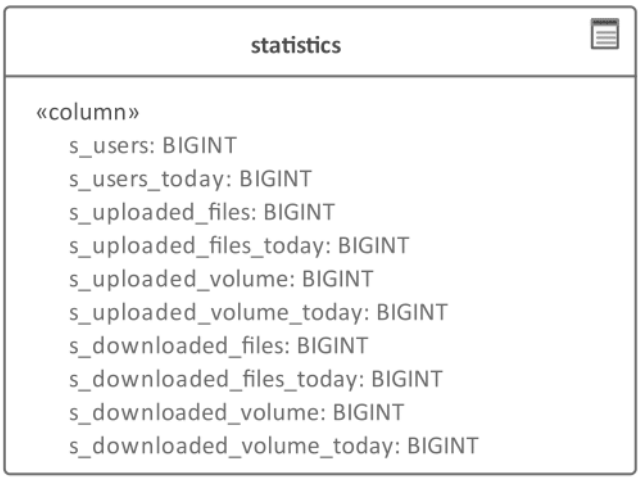


*Рисунок 5.3.c — **Таблица** `statistics`*

Эта таблица должна всегда содержать ровно одну строку, в каждом поле которой находятся соответствующие данные.

С точки зрения надёжности стоило бы каждый раз проверять, ровно ли одна строка находится в этой таблице (и если строк больше, то удалять их все и создавать одну новую, а если нет ни одной строки — просто создавать одну новую).

Но это катастрофически скажется на производительности, потому операцию по созданию одной единственной строки придётся возложить на скрипт по формированию базы данных, а в дальнейшем полагаться на то, что никто и ничто не удалит эту строку и не добавит новых строк.

Также на производительность сильно повлияет стратегия формирования данных, т.к. мы можем:

- каждый раз заново вычислять все необходимые значения (это очень просто, но очень медленно, т.к. придётся выполнять несколько запросов с `COUNT()` на таблицах с большим количеством записей);
- изменять имеющиеся значения на соответствующие величины (это намного быстрее, но потребует доработки самой таблицы).

Пойдём по второму пути и выполним доработку таблицы (см. рисунок 5.3.d). Её необходимость вызвана тем, что часть данных нужно считать «за сегодня», и потому нужно понимать, когда «сегодня» превращается во «вчера».

Добавим в эту таблицу поле **`s_actual_date`**, в котором будем хранить информацию о предполагаемой текущей дате. Затем каждый раз триггер будет определять настоящую текущую дату. И если её значение отличается от значения поля **`s_actual_date`**, значит, наступил новый день.

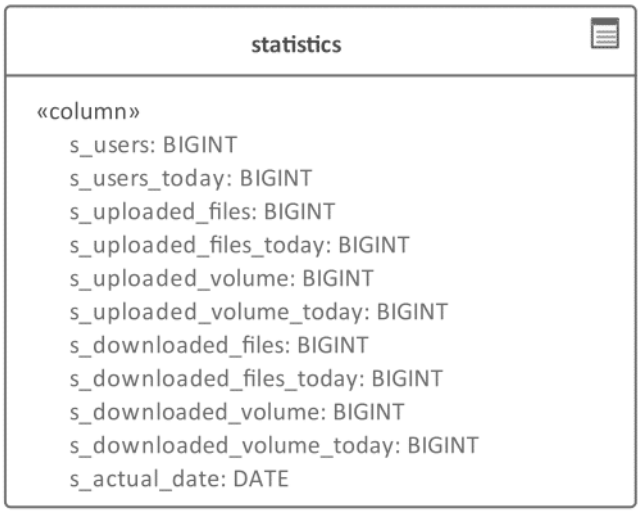


*Рисунок 5.3.d — **Таблица `statistics` после доработки***

Также выясняется (и очень хорошо иллюстрирует тот факт, что ошибки в проектировании базы данных могут быть выявлены на любом уровне и в любой момент), что существует недоработка в таблице **`file`** — там нет поля, отвечающего за количество скачиваний файла. Доработаем и эту таблицу (см. рисунок 5.3.e), добавив поле **`f_download_count`** и **`f_download_count_today`**.

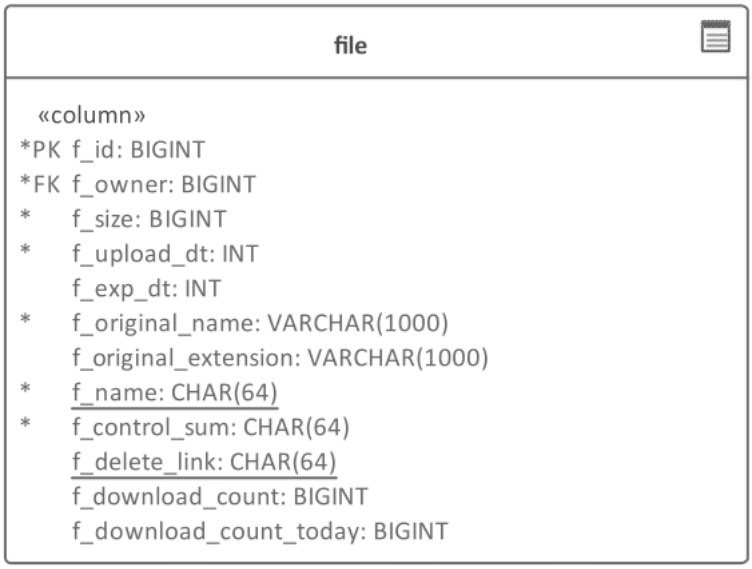


*Рисунок 5.3.e — Таблица **`file`** после доработки*

Теперь необходимо на таблице **file** создать три триггера (да, похожие триггеры нужно создать и на таблице **user**, но это — задача для самостоятельной проработки, см. задание 5.3.c{362}):

- **AFTER INSERT** — произошло добавление нового файла, нужно добавить информацию о нём в таблицу **statistics**;
- **AFTER DELETE** — произошло удаление файла, нужно убрать информацию о нём из таблицы **statistics**;
- **AFTER UPDATE** — произошло изменение информации о файле (возможно, увеличился счётчик скачиваний, и нужно добавить эту информацию в таблицу **statistics**).

Все три триггера — **AFTER**, т.е. они будут вызваны СУБД только после успешного завершения операции модификации данных в таблице **file**. Так мы исключаем ситуацию, когда СУБД в случае неудачной операции с таблицей **file** придётся также отменять все изменения, выполненные триггерами.

Также все три триггера должны будут контролировать факт смены дня (когда «сегодня» превратилось во «вчера») и выполнять некоторые дополнительные операции.

Мы реализуем нашу базу данных с использованием MySQL, а эта СУБД не позволяет один и тот же триггер ассоциировать с несколькими событиями (т.е. сделать так, чтобы один и тот же триггер выполнялся и при вставке, и при обновлении, и при удалении данных), потому у нас снова есть два варианта:

- дублировать часть кода во всех триггерах;
- перенести часть кода в хранимую процедуру{363} и вызывать её из триггеров.

Второй вариант намного более выгоден, т.к. позволяет использовать хранимую процедуру и вне триггеров.

Хранимым процедурам посвящена следующая глава{363}, но т.к. данная процедура нужна нам уже сейчас, с её кода и начнём.

MySQL **Хранимая процедура, очищающая статистику при наступлении нового дня**

```
1   DROP PROCEDURE IF EXISTS NEW_DAY;
2   DELIMITER $$
3   CREATE PROCEDURE NEW_DAY ()
4   BEGIN
5    IF EXISTS (SELECT 1 FROM `statistics`
6                       WHERE `s_actual_date` != CURRENT_DATE())
7     THEN
8      UPDATE `statistics` SET
9       `s_users_today` = 0,
10      `s_uploaded_files_today` = 0,
11      `s_uploaded_volume_today` = 0,
12      `s_downloaded_files_today` = 0,
13      `s_downloaded_volume_today` = 0,
14      `s_actual_date` = CURRENT_DATE();
15     UPDATE `file` SET
16      `f_download_count_today` = 0;
17    END IF;
18  END;
19  $$ DELIMITER ;
```

Запрос в строках 5-6 проверяет, есть ли в таблице **statistics** хотя бы одна строка, в поле **s_actual_date** которой хранится **не** сегодняшняя дата. Если условие выполнилось, значит наступил новый день и всю «статистику за сегодня» нужно обнулить, что и выполняют запросы в строках 8-14 и 15-16. В строке 14 также изменяется значение сегодняшней даты, хранимое в таблице **statistics**.

Эту процедуры мы будем вызывать в каждом из следующих трёх триггеров.

Начнём с триггера, срабатывающего после вставки данных.

MySQL **Триггер, срабатывающий при добавлении файла**

```
1   DROP TRIGGER IF EXISTS `TRG_update_file_stats_after_ins`;
2   DELIMITER $$
3
4   CREATE TRIGGER `TRG_update_file_stats_after_ins` AFTER INSERT ON `file`
5    FOR EACH ROW BEGIN
6     CALL NEW_DAY();
7     UPDATE `statistics` SET
8       `s_uploaded_files` = `s_uploaded_files` + 1,
9       `s_uploaded_files_today` = `s_uploaded_files_today` + 1,
10      `s_uploaded_volume` = `s_uploaded_volume` + NEW.`f_size`,
11      `s_uploaded_volume_today` = `s_uploaded_volume_today` + NEW.`f_size`;
12   END;
13  $$
14  DELIMITER ;
```

После вызова хранимой процедуры (строка 6), обнуляющей вчерашнюю статистику в случае смены дня, мы увеличиваем счётчики закачанных файлов (строки 8-9) и увеличиваем значение объёма закачанных файлов (строки 10-11). И всё.

Следующий триггер реагирует на обновление данных. В общем случае мы рассчитываем на то, что у файла изменился счётчик скачиваний. Но (по крайней мере в теории) у файла может измениться также и размер (и даже количество скачиваний произвольным образом). Потому код триггера будет немного сложнее.

MySQL **Триггер, срабатывающий при обновлении файла**

```
1   DROP TRIGGER IF EXISTS `TRG_update_file_stats_after_upd`;
2   DELIMITER $$
3
4   CREATE TRIGGER `TRG_update_file_stats_after_upd` AFTER UPDATE ON `file`
5    FOR EACH ROW BEGIN
6     IF (FROM_UNIXTIME(OLD.`f_upload_dt`) = CURRENT_DATE())
7      THEN
8       UPDATE `statistics` SET
9          `s_uploaded_volume_today` = `s_uploaded_volume_today` -
10                                      OLD.`f_size`;
11      UPDATE `statistics` SET
12         `s_uploaded_volume_today` = `s_uploaded_volume_today` +
13                                      NEW.`f_size`;
14      UPDATE `statistics` SET
15         `s_downloaded_files_today` = `s_downloaded_files_today` -
16                                       OLD.`f_download_count_today`;
17      UPDATE `statistics` SET
18         `s_downloaded_files_today` = `s_downloaded_files_today` +
19                                       NEW.`f_download_count_today`;
20      UPDATE `statistics` SET
21         `s_downloaded_volume_today` = `s_downloaded_volume_today` -
22                        OLD.`f_size` * OLD.`f_download_count_today`;
23      UPDATE `statistics` SET
24         `s_downloaded_volume_today` = `s_downloaded_volume_today` +
25                        NEW.`f_size` * NEW.`f_download_count_today`;
26     END IF;
27
28      CALL NEW_DAY();
29      UPDATE `statistics` SET
30         `s_uploaded_volume` = `s_uploaded_volume` - OLD.`f_size`;
31      UPDATE `statistics` SET
32         `s_uploaded_volume` = `s_uploaded_volume` + NEW.`f_size`;
33      UPDATE `statistics` SET
34         `s_downloaded_files` = `s_downloaded_files` -
35                                OLD.`f_download_count`;
```

```
36      UPDATE `statistics` SET
37         `s_downloaded_files` = `s_downloaded_files` +
38                                NEW.`f_download_count`;
39      UPDATE `statistics` SET
40         `s_downloaded_volume` = `s_downloaded_volume` -
41                 OLD.`f_size` * OLD.`f_download_count`;
42      UPDATE `statistics` SET
43         `s_downloaded_volume` = `s_downloaded_volume` +
44                 NEW.`f_size` * NEW.`f_download_count`;
45
46   END;
47  $$
48  DELIMITER ;
```

В данном случае необходимо проверить, сегодня ли был загружен файл (строка 6) и, если да, подкорректировать сегодняшнюю статистику (строки 8-25). После чего можно вызывать хранимую процедуру (строка 28), обнуляющую вчерашнюю статистику в случае смены дня, и корректировать общую статистику (строки 29-44).

И, наконец, остался триггер, реагирующий на удаление данных. Он получается практически идентичным предыдущему за тем лишь исключением, что здесь статистические данные будут корректироваться только в сторону уменьшения.

MySQL **Триггер, срабатывающий при добавлении файла**

```
1   DROP TRIGGER IF EXISTS `TRG_update_file_stats_after_del`;
2   DELIMITER $$
3
4   CREATE TRIGGER `TRG_update_file_stats_after_del` AFTER DELETE ON `file`
5    FOR EACH ROW BEGIN
6     IF (FROM_UNIXTIME(OLD.`f_upload_dt`) = CURRENT_DATE())
7      THEN
8       UPDATE `statistics` SET
9          `s_uploaded_volume_today` = `s_uploaded_volume_today` -
10                                     OLD.`f_size`;
11      UPDATE `statistics` SET
12         `s_uploaded_files_today` = `s_uploaded_files_today` - 1;
13      UPDATE `statistics` SET
14         `s_downloaded_files_today` = `s_downloaded_files_today` -
15                                      OLD.`f_download_count_today`;
16      UPDATE `statistics` SET
17         `s_downloaded_volume_today` = `s_downloaded_volume_today` -
18                        OLD.`f_size` * OLD.`f_download_count_today`;
19     END IF;
20
21      CALL NEW_DAY();
22      UPDATE `statistics` SET
23         `s_uploaded_volume` = `s_uploaded_volume` - OLD.`f_size`;
24      UPDATE `statistics` SET
25         `s_uploaded_files` = `s_uploaded_files` - 1;
26      UPDATE `statistics` SET
27         `s_downloaded_files` = `s_downloaded_files` -
28                                OLD.`f_download_count`;
29      UPDATE `statistics` SET
30         `s_downloaded_volume` = `s_downloaded_volume` -
31                 OLD.`f_size` * OLD.`f_download_count`;
32
33   END;
34  $$
35  DELIMITER ;
```

Ещё раз отметим, что для того, чтобы таблица **statistics** обновлялась полностью, похожие триггеры нужно создать и на таблице **user**, но это — задача для самостоятельной проработки, см. задание 5.3.c{362}).

Мы же переходим ко второму примеру — контролю с помощью триггеров бизнес-правила «в системе всегда должен быть хотя бы один администратор».

Это правило может быть нарушено в двух случаях:

- происходит попытка удалить единственного пользователя, находящегося в группе «администраторы»;
- происходит попытка убрать из группы «администраторы» единственного находящегося в ней пользователя.

В обеих ситуациях мы должны выполнить соответствующие проверки и запретить операцию, если её выполнение приведёт к нарушению бизнес-правила.

Некоторая сложность состоит в том, что не существует строгого формального способа определить группу «администраторы». Мы можем полагаться только на её название или исходить из предположения о том, что её идентификатор фиксирован, известен нам и не будет меняться в будущем.

Второй подход более оправдан, т.к. в таблице **group** первичный ключ является искусственным{42}, а потому едва ли будет меняться. И мы также можем на стадии формирования базы данных добавить в скрипт операцию по созданию соответствующей группы пользователей с заданным идентификатором.

Таким образом, будем считать, что группа «администраторы» описана в таблице **group** записью со значением первичного ключа равным 1.

Напишем триггер, запрещающий удалять единственного пользователя, находящегося в группе «администраторы». Это будет триггер вида **BEFORE DELETE** на таблице **user**.

MySQL **Триггер, запрещающий удалять последнего администратора**

```
1   DROP TRIGGER IF EXISTS `TRG_preserve_last_admin_before_del`;
2   DELIMITER $$
3
4   CREATE TRIGGER `TRG_preserve_last_admin_before_del` BEFORE DELETE ON `user`
5    FOR EACH ROW BEGIN
6     IF (NOT EXISTS (SELECT 1
7                     FROM   `m2m_user_group`
8                     WHERE  (`ug_group` = 1)
9                     AND (`ug_user` != OLD.`u_id`)))
10     THEN
11      SIGNAL SQLSTATE '45001'
12      SET MESSAGE_TEXT = 'You can not delete the last user from
13                          Administrators (`g_id` = 1) group.',
14      MYSQL_ERRNO = 1001;
15     END IF;
16   END;
17   $$
18   DELIMITER ;
```

В строках 6-9 выполняется проверка наличия в группе «администраторы» хотя бы одного пользователя с идентификатором, отличным от идентификаторы удаляемого пользователя. Если таковых не нашлось, в строках 11-14 создаётся исключительная ситуация, блокирующая операцию и отменяющая текущую транзакцию.

Обратите внимание, что в отличие от проверок{347}, здесь мы можем сформировать информативное сообщение об ошибке. И при попытке удалить последнего пользователя, находящегося в группе «администраторы» оно будет отображено:

MySQL **Сообщение об ошибке**

```
Error Code: 1001.
You can not delete the last user from Administrators (`g_id` = 1) group.
```

Теперь необходимо создать два триггера (**BEFORE DELETE** и **BEFORE UPDATE**) на таблице **m2m_user_group**, запрещающие убирать из группы «администраторы» последнего находящегося там пользователя.

Их код будет полностью идентичным, и если бы MySQL позволял ассоциировать один триггер с несколькими операциями, можно было бы обойтись одним триггером. Но такой возможности в MySQL пока нет, потому придётся создавать два отдельных триггера.

MySQL **Триггер, запрещающий удалять из группы «администраторы» последнего пользователя**

```
DROP TRIGGER IF EXISTS `TRG_preserve_last_user_in_admins_before_del`;
DELIMITER $$

CREATE TRIGGER `TRG_preserve_last_user_in_admins_before_del` BEFORE DELETE
ON `m2m_user_group`
 FOR EACH ROW BEGIN
  IF (NOT EXISTS (SELECT 1
                  FROM   `m2m_user_group`
                  WHERE  (`ug_group` = 1)
                  AND (`ug_user` != OLD.`ug_user`)))
   THEN
    SIGNAL SQLSTATE '45001'
    SET MESSAGE_TEXT = 'You can not remove the last user from
                        Administrators (`g_id` = 1) group.',
    MYSQL_ERRNO = 1001;
  END IF;
 END;
$$
DELIMITER ;
```

MySQL **Триггер, запрещающий перемещать из группы «администраторы» последнего пользователя**

```
DROP TRIGGER IF EXISTS `TRG_preserve_last_user_in_admins_before_upd`;
DELIMITER $$

CREATE TRIGGER `TRG_preserve_last_user_in_admins_before_upd` BEFORE UPDATE
ON `m2m_user_group`
 FOR EACH ROW BEGIN
  IF (NOT EXISTS (SELECT 1
                  FROM   `m2m_user_group`
                  WHERE  (`ug_group` = 1)
                  AND (`ug_user` != OLD.`ug_user`)))
   THEN
    SIGNAL SQLSTATE '45001'
    SET MESSAGE_TEXT = 'You can not remove the last user from
                        Administrators (`g_id` = 1) group.',
    MYSQL_ERRNO = 1001;
  END IF;
 END;
$$
DELIMITER ;
```

В строках 7-10 обоих запросов выполняется проверка наличия в группе «администраторы» хотя бы одного пользователя с идентификатором, отличным от идентификатора удаляемого (или перемещаемого) из группы пользователя. Если таковых не нашлось, в строках 12-15 обоих запросов создаётся исключительная ситуация, блокирующая операцию и отменяющая текущую транзакцию.

И при попытке удалить или переместить из группы «администраторы» последнего находящегося там пользователя, будет отображено:

MySQL **Сообщение об ошибке**

```
1   Error Code: 1001.
2   You can not remove the last user from Administrators (`g_id` = 1) group.
```

Множество более сложных практических примеров приведено в разделе 4 «Использование триггеров» книги131 «Работа с MySQL, MS SQL Server и Oracle в примерах».

Задание 5.3.b: создайте триггеры, список которых вы составили при выполнении задания 5.3.a{354}.

Задание 5.3.c: ранее{357} в данной главе был рассмотрен пример создания на таблице **`file`** всех необходимых для обновления статистики триггеров. И было отмечено, что похожие триггеры нужно создать и на таблице **`user`**. Создайте соответствующие триггеры.

221 «Работа с MySQL, MS SQL Server и Oracle в примерах» (С.С. Куликов) [http://svyatoslav.biz/database_book/]

5.4. ХРАНИМЫЕ ПОДПРОГРАММЫ

5.4.1. ОБЩИЕ СВЕДЕНИЯ О ХРАНИМЫХ ПОДПРОГРАММАХ

Как и триггеры{350}, хранимые подпрограммы очень по-разному реализованы в различных СУБД, и речь снова идёт не только о синтаксисе, но и о фундаментальных отличиях и возможностях (какие операции могут выполнять хранимые подпрограммы, как эти подпрограммы можно использовать и т.д.).

Начнём с определений (это особенно важно для понимания разницы между хранимыми процедурами и хранимыми функциями (которые также иногда называют «пользовательскими функциями»)).

Хранимая процедура (stored procedure[222]) — подпрограмма (возможно, параметризованная) предназначенная для выполнения ряда операций с данными и структурами базы данных, хранимая на стороне базы данных и доступная как для вызова из кода других процедур и триггеров, так и для непосредственного исполнения.

Упрощённо: подпрограмма, вызываемая напрямую или из других подпрограмм, и выполняющая некоторые полезные действия.

Хранимая функция, пользовательская функция (stored function, user-defined function[223]) — подпрограмма (возможно, параметризованная), расширяющая возможности языка SQL и работающая аналогично встроенным в СУБД функциям; обязана возвращать значения.

Упрощённо: функция, расширяющая возможности СУБД и предназначенная для упрощения часто повторяющихся операций.

Итак, первое и самое главное отличие: функция обязана возвращать значение, процедура — не обязана (и часто даже не имеет такой возможности). Второе (типичное для большинства СУБД) отличие состоит в том, что функцию можно использовать в любом SQL-запросе, в то время как работа с процедурами имеет свой особый синтаксис и ряд ограничений.

---

222 **Stored procedure** — a subroutine, possibly parameterized; in other words, the implementation code for some operator. The term «stored procedure» has unfortunately come to mean something in practice that mixes model and implementation considerations. From the point of view of the model, a stored procedure is basically nothing more than an operator (or the implementation code for such an operator, rather). In practice, however, stored procedures have a number of properties that make them much more important than they would be if they were just operators as such (although the first two of the following properties will probably apply to operators in general, at least if the operators in question are system defined). First, they're compiled separately and can be shared by distinct applications. Second, their compiled code is, typically, physically stored at the site at which the data itself is physically stored, with obvious performance benefits. Third, they're often used to provide shared functionality that ought to be provided by the DBMS but isn't (integrity checking is a good example here, given the state of today's SQL implementations). («The New Relational Database Dictionary», C.J. Date)

223 **Stored function, user-defined function** — a way to extend SQL with a new function that works like a native (built-in) SQL function such as ABS() or CONCAT(). [https://dev.mysql.com/doc/refman/8.0/en/create-function-udf.html]

Поскольку на этом отличия не заканчиваются, приведём их в виде таблицы (эти утверждения справедливы для большинства СУБД).

Хранимая процедура	Хранимая функция
Может иметь несколько входных и выходных параметров	Может иметь несколько входных параметров, и всегда обязана возвращать значение (в некоторых СУБД это значение может представлять собой таблицу)
Имеет свой собственный синтаксис для вызова и обработки результатов	Может быть использована в любом SQL-запросе
Может вызывать другие хранимые процедуры и функции	Может вызывать только другие хранимые функции, но не хранимые процедуры
Может порождать транзакции	Не может порождать транзакции
Обладает полным спектром возможностей по обработке исключительных ситуаций	Обладает очень узкими возможностями по обработке исключительных ситуаций (в некоторых СУБД — вообще не может обрабатывать такие ситуации)
Может выполнять любые операции с данными	Может выполнять только чтение данных
Может выполнять любые операции с объектами базы данных	Может выполнять только чтение информации об объектах базы данных
Не может быть использована в проверке{347}	Может быть использована в проверке{347}

Если свести все отличия к одной фразе, получается, что функции мы используем тогда, когда в SQL-запросе нужно получить некоторое значение (по аналогии со встроенными функциями наподобие `SUM()`, `AVG()` и т.д.), а процедуры — когда нужно выполнить ряд сложных действий (обновить данные, создать новую таблицу и т.д.)

Проиллюстрируем это графически (см. рисунок 5.4.а).

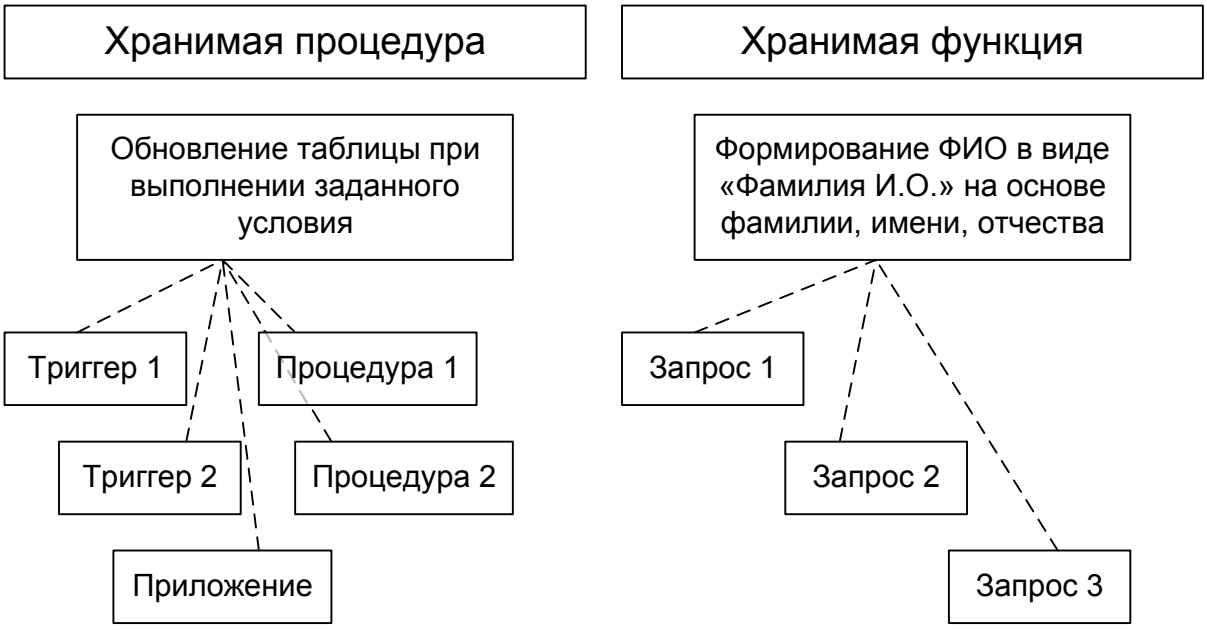


*Рисунок 5.4.а — **Пример работы хранимых процедур и хранимых функций***

В обоих случаях достигается следующий ряд преимуществ:

- необходимые действия описаны один раз, т.е. не нужно писать один и тот же код многократно в разных местах;
- вызов подпрограммы — это короткая синтаксическая конструкция, что намного упрощает читаемость кода (в сравнении с ситуацией, когда все соответствующие действия были бы описаны явно);
- подпрограмма работает на стороне СУБД, т.е. никакие данные не передаются в приложение или куда бы то ни было ещё, что повышает производительность и безопасность.

Недостатки хранимых подпрограмм не столь очевидны и очень сильно зависят от конкретной СУБД, потому следующие утверждения носят «возможный» характер:

- в некоторых случаях возможно снижение производительности (по сравнению с непосредственным выполнением кода, перенесённого в хранимую подпрограмму);
- в некоторых случаях возможны проблемы с безопасностью (если на выполнение хранимой подпрограммы выставлены неверные права);
- в некоторых случаях использование хранимых подпрограмм может усложнить тестирование и диагностику неполадок.

Однако в общем и целом преимущества хранимых подпрограмм сильно превышают их недостатки.

И ещё раз подчеркнём, что синтаксис, особенности реализации, возможности и ограничения хранимых подпрограмм очень сильно отличаются в различных СУБД (и даже в различных версиях одной и той же СУБД), потому обязательно изучайте документацию, экспериментируйте и тщательно тестируйте написанный код.

Задание 5.4.а: составьте список хранимых подпрограмм, которые стоило бы добавить в базу данных «Банк»{408}. Отметьте для каждой такой подпрограммы, какие задачи решаются с её помощью.

5.4.2. СОЗДАНИЕ И ИСПОЛЬЗОВАНИЕ ХРАНИМЫХ ПОДПРОГРАММ

В качестве примера рассмотрим решение следующих задач:
- корректировка статистических данных при смене дня;
- удаление информации с истёкшим сроком хранения;
- определение статуса пользователя («новичок», «опытный», «мастер») в зависимости от количества и объёма закачанных и скачанных файлов;
- формирование информации о размере файла в удобном для восприятия человеком виде с подстройкой размерности (например, «10.35 MB», «1.24 GB» и т.д.).

Первые две задачи будут реализованы с использованием хранимых процедур, оставшиеся — с использованием хранимых (пользовательских) функций.

Абсолютное большинство средств проектирования позволяет добавить в модель базы данных отдельный объект, описывающий хранимую процедуру или функцию. В используемом нами средстве Sparx Enterprise Architect эти объекты называются «Procedure» и «Function» соответственно.

Корректировка статистических данных при смене дня уже была рассмотрена нами ранее{357}, но сейчас мы изучим этот пример подробнее. Начнём с кода хранимой процедуры.

MySQL **Хранимая процедура, очищающая статистику при наступлении нового дня**

```
1   DROP PROCEDURE IF EXISTS NEW_DAY;
2   DELIMITER $$
3   CREATE PROCEDURE NEW_DAY ()
4   BEGIN
5    IF EXISTS (SELECT 1 FROM `statistics`
6                       WHERE `s_actual_date` != CURRENT_DATE())
7     THEN
8      UPDATE `statistics` SET
9       `s_users_today` = 0,
10      `s_uploaded_files_today` = 0,
11      `s_uploaded_volume_today` = 0,
12      `s_downloaded_files_today` = 0,
13      `s_downloaded_volume_today` = 0,
14      `s_actual_date` = CURRENT_DATE();
15     UPDATE `file` SET
16      `f_download_count_today` = 0;
17    END IF;
18  END;
19  $$ DELIMITER ;
```

Напомним, что запрос в строках 5-6 проверяет, есть ли в таблице **statistics** хотя бы одна строка, в поле **s_actual_date** которой хранится НЕ сегодняшняя дата. Если условие выполнилось, значит наступил новый день и всю «статистику за сегодня» нужно обнулить, что и выполняют запросы в строках 8-14 и 15-16. В строке 14 также изменяется значение сегодняшней даты, хранимое в таблице **statistics**.

Мы использовали эту хранимую процедуру в нескольких триггерах, но есть и более элегантное решение, позволяющее исключить вызов этой процедуры при каждом срабатывании каждого триггера, что очевидно повысит производительность этих триггеров.

Многие СУБД (MySQL в их числе) поддерживают возможность выполнения заданных действий по расписанию. К таким действиям относится и выполнение хранимой процедуры. К сожалению, многие средства проектирования не позволяют добавлять такую информацию в модель базы данных, потому соответствующий SQL-код необходимо поместить вручную в скрипт по формированию базы данных.

Поскольку по умолчанию планировщик событий (отвечающий за выполнение действий по расписанию) в MySQL может быть выключен, необходимо в конфигурационный файл my.ini добавить в группу опций `[mysqld]` такую строку:

```
[event_scheduler = ON]
```

После чего перезапустить сервис MySQL.

Возвращаемся к сути задачи. Смена дня объективно происходит в полночь, потому логично выполнять все соответствующие действия в 0 часов 0 минут 0 секунд каждого дня. Чтобы добавить в расписание соответствующее событие, достаточно выполнить следующий SQL-запрос:

MySQL **Создание события для запуска хранимой процедуры NEW_DAY каждую полночь**

```
1   CREATE EVENT `clear_statistics_at_midnight`
2    ON SCHEDULE
3     EVERY 1 DAY
4     STARTS '2000-01-01 00:00:00' ON COMPLETION PRESERVE ENABLE
5    DO
6     CALL NEW_DAY()
```

Выражение `STARTS` в 4-й строке запроса (дату лучше поставить в прошлом) указывает точку отсчёта (в ней важно именно время, т.е. 00:00:00). Далее это событие будет повторяться каждый день (строка 3 запроса).

Теперь соответствующая хранимая процедура будет автоматически запускаться каждую полночь и корректировать статистические данные. И это значит, что можно переписать ранее показанные{357} триггеры таким образом, чтобы исключить необходимость вызова хранимой процедуры из их кода (см. задание 5.4.c{373}).

Переходим ко второй задаче, т.е. к удалению информации с истёкшим сроком хранения.

Во многих таблицах нашей базы данных есть поля, имена которых заканчиваются на `exp_dt`:

- user
  - u_speed_bonus_exp_dt
  - u_volume_bonus_exp_dt
  - u_ban_exp_dt
- ip_blacklist
  - ibl_exp_dt
- file
  - f_exp_dt
- download_link
  - dl_exp_dt

Эти поля содержат данные о сроке хранения соответствующей информации. По истечению этого срока (наступлению указанных даты и времени) необходимо удалить или изменить информацию в базе данных, а именно:

- Таблица user:
  - u_speed_bonus_exp_dt — удалить информацию о бонусе в виде скорости по количеству закачанного;
  - u_volume_bonus_exp_dt — удалить информацию о бонусе в виде объёма по количеству закачанного;
  - u_ban_exp_dt — удалить информацию о блокировке;

- Таблица ip_blacklist:
  - ibl_exp_dt — удалить ip-адрес из списка заблокированных;
- Таблица file:
  - f_exp_dt — удалить файл;
- Таблица download_link:
  - dl_exp_dt — удалить ссылку для скачивания файла.

И такие изменения в базе данных должны происходить полностью автоматически. Легко догадаться, что мы используем тот же подход, что и в предыдущем примере — создадим хранимую процедуру и добавим её выполнение в расписание планировщика событий. Остаётся лишь вопрос о частоте вызова этой хранимой процедуры: чем чаще она будет вызываться, тем ближе к реальному времени будут изменяться данные, но и тем больше будет нагрузка на базу данных.

Здесь нет «единого правильного решения», всегда необходимо исходить из реальной ситуации, но предположим, что нас (и заказчика) устроит вариант, когда такие изменения будут происходить раз в 30 минут.

Начнём с кода хранимой процедуры:

MySQL **Хранимая процедура, удаляющая устаревшую информацию**

```
DROP PROCEDURE IF EXISTS CLEAR_OUTDATED_OBJECTS;
DELIMITER $$
CREATE PROCEDURE CLEAR_OUTDATED_OBJECTS ()
BEGIN

 UPDATE `user` SET `u_speed_bunus` = NULL
  WHERE `u_speed_bunus_exp_dt` < UNIX_TIMESTAMP();

 UPDATE `user` SET `u_volume_bunus` = NULL
  WHERE `u_volume_bunus_exp_dt` < UNIX_TIMESTAMP();

 UPDATE `user` SET `u_ban` = NULL
  WHERE `u_ban_exp_dt` < UNIX_TIMESTAMP();

 DELETE FROM `ip_blacklist`
  WHERE `ibl_exp_dt` < UNIX_TIMESTAMP();

 DELETE FROM `file`
  WHERE `f_exp_dt` < UNIX_TIMESTAMP();

 DELETE FROM `download_link`
  WHERE `dl_exp_dt` < UNIX_TIMESTAMP();

END;
$$ DELIMITER ;
```

Каждый из шести запросов выставляет в `NULL` значение внешнего ключа или удаляет соответствующую запись, если истёк срок хранения указанной информации.

Теперь необходимо сделать так, чтобы созданная хранимая процедура автоматически выполнялась раз в 30 минут. Для этого добавим событие:

MySQL **Создание события для запуска хранимой процедуры CLEAR_OUTDATED_OBJECTS раз в 30 минут**

```
CREATE EVENT `clear_outdated_objects_every_30_minutes`
 ON SCHEDULE
  EVERY 30 MINUTE
  STARTS '2000-01-01 00:01:00' ON COMPLETION PRESERVE ENABLE
 DO
  CALL CLEAR_OUTDATED_OBJECTS()
```

Выражение **STARTS** в 4-й строке запроса (дату лучше поставить в прошлом) указывает точку отсчёта (т.к. ровно в полночь у нас уже выполняется процедура **NEW_DAY**, здесь укажем небольшое смещение, т.е. 00:01:00, чтобы две процедуры не выполнялись параллельно и не снижали тем самым производительность). Далее это событие будет повторяться каждые 30 минут (строка 3 запроса).

Переходим к функциям.

В первой из задач нам нужно определить статус пользователя («новичок», «опытный», «мастер») в зависимости от количества и объёма закачанных и скачанных файлов.

Предположим, что от заказчика мы получили следующую информацию о том, как определяется такой статус (при этом важно, что учитывается «большее из достижений», т.е. если пользователь закачал, например, всего один файл объёмом более 10 GB, он всё равно получает статус «мастер»):

Статус	Объём закачанных файлов	Количество закачанных файлов
Новичок	< 1 GB	< 100
Опытный	1-10 GB	100-1000
Мастер	> 10 GB	> 1000

Алгоритм определения статуса будет следующим:

- определить статус по объёму закачанных файлов;
- определить статус по количеству закачанных файлов;
- выбрать больший из этих двух статусов;

Также полезно будет добавить возможность возвращать результат в виде числа или в виде строковой константы. Да, «число» здесь тоже будет строковой константой, но его всегда можно в дальнейшем преобразовать к «настоящему числу».

Итак, вот код функции:

MySQL **Функция, определяющая статус пользователя**

```
1   DROP FUNCTION IF EXISTS GET_USER_STATUS;
2   DELIMITER $$
3   CREATE FUNCTION GET_USER_STATUS(uploaded_volume BIGINT UNSIGNED,
4                                   uploaded_count BIGINT UNSIGNED,
5                                   return_mode VARCHAR(10))
6   RETURNS VARCHAR(150) DETERMINISTIC
7   BEGIN
8    DECLARE uploaded_volume_status INT;
9    DECLARE uploaded_count_status INT;
10   DECLARE final_status INT;
11
12   CASE
13    WHEN (uploaded_volume < 1073741824) THEN SET uploaded_volume_status = 1;
14    WHEN ((uploaded_volume >= 1073741824)
15      AND (uploaded_volume <= 10737418240)) THEN
16                                          SET uploaded_volume_status = 2;
17    WHEN (uploaded_volume > 10737418240) THEN SET uploaded_volume_status = 3;
18   END CASE;
19
20   CASE
21    WHEN (uploaded_count < 100) THEN SET uploaded_count_status = 1;
22    WHEN ((uploaded_count >= 100)
23      AND (uploaded_count <= 1000)) THEN SET uploaded_count_status = 2;
24    WHEN (uploaded_count > 1000) THEN SET uploaded_count_status = 3;
25   END CASE;
26
27
```

```
28   SET final_status = (SELECT GREATEST(uploaded_volume_status,
29                                       uploaded_count_status));
30
31   IF (return_mode = 'NUMBER')
32    THEN
33     RETURN CONCAT(final_status, '');
34    ELSE
35     CASE
36      WHEN (final_status = 1) THEN RETURN 'NOVICE';
37      WHEN (final_status = 2) THEN RETURN 'EXPERIENCED';
38      WHEN (final_status = 3) THEN RETURN 'MASTER';
39     END CASE;
40   END IF;
41
42  END;
43  $$
44
45  DELIMITER ;
```

В строках 12-18 происходит определение статуса по объёму закачанных файлов, в строках 20-25 происходит определение статуса по количеству закачанных файлов, в строках 28-29 происходит выбор большего из этих двух статусов, и в строках 31-40 происходит определение того, как именно (в виде числа или строки) нужно возвращать результат, а также непосредственно возвращение результата.

Выражение **`CONCAT(final_status, '')`** в строке 33 позволяет преобразовать число к строковой форме представления, т.к. данная функция обязана возвращать **`VARCHAR(150)`** (о чём сказано в строке 6).

Ключевое слово **`DETERMINISTIC`** в строке 6 говорит о том, что при получении на вход одних и тех же значений данная функция всегда будет возвращать один и тот же результат.

Использование этой функции потребует предварительной подготовки данных. Технически мы могли бы передавать внутрь функции всего лишь идентификатор пользователя и выполнять все необходимые запросы внутри, но это губительно сказалось бы на производительности[224].

Итак, вот пример запроса, в котором происходит использование данной функции:

MySQL **Пример запроса для использования функции, определяющей статус пользователя**

```
1   SELECT `u_id`,
2          `u_login`,
3          SUM(`f_size`) AS `files_volume`,
4          COUNT(`f_id`) AS `files_count`,
5          GET_USER_STATUS(SUM(`f_size`),
6                          COUNT(`f_id`),
7                          'NUMBER') AS `user_status`
8     FROM `user`
9     JOIN `file`
10      ON `u_id` = `f_owner`
11   GROUP BY (`u_id`)
```

Строки 3 и 4 здесь не обязательны и представлены просто для наглядности. Аналогичные вычисления производятся в строках 5 и 6 (и результат сразу передаётся как аргумент функции **`GET_USER_STATUS`**).

Ещё раз посмотрите на объём кода функции и размер запроса с её использованием. И представьте, что все аналогичные вычисления пришлось бы вписать непосредственно в запрос (к слову, синтаксис такого решения был бы ещё более сложным). Этот пример очень наглядно демонстрирует пользу применения хранимых функций.

---

[224] При небольшом «тестовом эксперименте» разница в производительности на 500’000 пользователей и 10’000’000 файлов составила более 100’000 раз. Т.е. разница в производительности — пять порядков.

И у нас осталась последняя задача: формирование информации о размере файла в удобном для восприятия человеком виде с подстройкой размерности (например, «10.35 MB», «1.24 GB» и т.д.).

Алгоритм решения здесь таков:

- определить диапазон, в который попадает размер файла (байты, килобайты, мегабайты и т.д.)
- округлить результат до сотых;
- добавить размерность.

Для максимального удобства мы можем сразу реализовать вычисление размера в единицах, кратных степени 10 (KB, MB, GB и т.д.) или в единицах, кратных степени 2 (KiB, MiB, GiB и т.д.).

Код функции выглядит так:

MySQL **Функция для формирования информации о размере файла в удобном для человека виде**

```
DROP FUNCTION IF EXISTS NORMALIZE_SIZE;
DELIMITER $$
CREATE FUNCTION NORMALIZE_SIZE(size BIGINT UNSIGNED,
                               measurement VARCHAR(10))
RETURNS VARCHAR(150) DETERMINISTIC
BEGIN
 DECLARE labels_2 VARCHAR(150)
  DEFAULT '["B","KiB","MiB","GiB","TiB","PiB","EiB","ZiB","YiB"]';
 DECLARE labels_10 VARCHAR(150)
  DEFAULT '["B","KB","MB","GB","TB","PB","EB","ZB","YB"]';

 DECLARE position_in_array_2 INT DEFAULT 0;
 DECLARE position_in_array_10 INT DEFAULT 0;

 DECLARE result_2 DOUBLE DEFAULT 0.0;
 DECLARE result_10 DOUBLE DEFAULT 0.0;

 SET position_in_array_2 = TRUNCATE(LOG(2, size) / LOG(2, 1024), 0);
 SET position_in_array_10 = TRUNCATE(LOG(10, size) / LOG(10, 1000), 0);

 SET result_2 = ROUND(size/POWER(1024, position_in_array_2), 2);
 SET result_10 = ROUND(size/POWER(1000, position_in_array_10), 2);

 IF (measurement = '2')
  THEN
   RETURN REPLACE(CONCAT(result_2, ' ', JSON_EXTRACT(labels_2,
                  CONCAT('$[',position_in_array_2,']'))), '"', '');
  ELSE
   RETURN REPLACE(CONCAT(result_10, ' ', JSON_EXTRACT(labels_10,
                  CONCAT('$[',position_in_array_10,']'))), '"', '');
 END IF;

END;
$$

DELIMITER ;
```

При написании этого кода использованы некоторые решения, выходящие за рамки данной книги, потому они будут лишь упомянуты, а мы сконцентрируемся на том, что здесь происходит с точки зрения SQL.

В строках 7-10 формируется два JSON-документа[225] — это самый простой и быстрый способ эмуляции массива данных в MySQL. В каждом из этих документов расположен набор надписей, обозначающих размеры в единицах, кратных степени 2 и кратных степени 10.

В строках 18-19 определяется, «в какой диапазон» попадает значение размера (байты, килобайты и т.д.).

В строках 21-22 определяется финальное значение размера в соответствующих единицах измерения.

В строках 24-31 подготавливается и возвращается итоговый результат.

Здесь функция **JSON_EXTRACT** позволяет извлечь из ранее сформированного JSON-документа элемент в указанной позиции. Второй аргумент этой функции формируется выражением **CONCAT('$[',position_in_array_10,']')**, где **position_in_array_10** — ранее определённый диапазон значений размера.

Применение функции **REPLACE** необходимо, чтобы убрать двойные кавычки (знак **"**), которыми будет обрамлён возвращённый функцией **JSON_EXTRACT** результат.

Проверим работоспособность созданной функции. Выполним следующий SQL-запрос:

MySQL **SQL-запрос для проверки работоспособности функции**

```
1   SELECT NORMALIZE_SIZE(1, '2'),
2          NORMALIZE_SIZE (1, '10')
3   UNION
4   SELECT NORMALIZE_SIZE (100, '2'),
5          NORMALIZE_SIZE (100, '10')
6   UNION
7   SELECT NORMALIZE_SIZE (1000, '2'),
8          NORMALIZE_SIZE (1000, '10')
9   UNION
10  SELECT NORMALIZE_SIZE (10000, '2'),
11         NORMALIZE_SIZE (10000, '10')
12  UNION
13  SELECT NORMALIZE_SIZE (1000000, '2'),
14         NORMALIZE_SIZE (1000000, '10')
15  UNION
16  SELECT NORMALIZE_SIZE (250000000, '2'),
17         NORMALIZE_SIZE (250000000, '10')
18  UNION
19  SELECT NORMALIZE_SIZE (47000000000, '2'),
20         NORMALIZE_SIZE (47000000000, '10')
21  UNION
22  SELECT NORMALIZE_SIZE (9800000000000, '2'),
23         NORMALIZE_SIZE (9800000000000, '10')
24  UNION
25  SELECT NORMALIZE_SIZE (7100000000000000, '2'),
26         NORMALIZE_SIZE (7100000000000000, '10')
27  UNION
28  SELECT NORMALIZE_SIZE (5340000000000000000, '2'),
29         NORMALIZE_SIZE (5340000000000000000, '10')
```

[225] **JSON** — an open standard file format, and data interchange format, that uses human-readable text to store and transmit data objects consisting of attribute-value pairs and array data types (or any other serializable value). («Wikipedia») [https://en.wikipedia.org/wiki/JSON]

В результате получим следующее:

В байтах	В значениях, кратных степени 2	В значениях, кратных степени 10
1	1 B	1 B
100	100 B	100 B
1000	1000 B	1 KB
1000000	9.77 KiB	10 KB
1000000	976.56 KiB	1 MB
250000000	238.42 MiB	250 MB
47000000000	43.77 GiB	47 GB
9800000000000	8.91 TiB	9.8 TB
7100000000000000	6.31 PiB	7.1 PB
5340000000000000000	4.63 EiB	5.34 EB

Для упрощения кода в созданной функции осознанно не добавлены никакие проверки, т.е. её поведение в некоторых ситуациях может быть ошибочным. Добавлению таких проверок и посвящено задание для самостоятельной проработки 5.4.d{373}.

Множество более сложных практических примеров приведено в разделе 5 «Использование хранимых функций и процедур» книги[226] «Работа с MySQL, MS SQL Server и Oracle в примерах».

Задание 5.4.b: создайте хранимые подпрограммы, список которых вы составили при выполнении задания 5.4.a{365}.

Задание 5.4.c: ранее{367} в данной главе было отмечено, что после автоматизации запуска хранимой процедуры NEW_DAY отпала необходимость её вызова из рассмотренных в предыдущем разделе триггеров. Переработайте код соответствующих триггеров так, чтобы исключить вызов из них указанной хранимой процедуры.

Задание 5.4.d: ранее{371} в данной главе при создании функции для формирования информации о размере файла в удобном для восприятия человеком виде мы осознанно (для упрощения кода) не создавали никаких проверок о ограничений, т.е. поведение этой функции в некоторых ситуациях может быть ошибочным. Добавьте в код функции соответствующие проверки и ограничения.

226 «Работа с MySQL, MS SQL Server и Oracle в примерах» (С.С. Куликов) [http://svyatoslav.biz/database_book/]

5.5. ТРАНЗАКЦИИ

5.5.1. ОБЩИЕ СВЕДЕНИЯ О ТРАНЗАКЦИЯХ

До сих пор мы говорили о разнообразных объектах баз данных. Транзакции же относятся к другой категории — это процессы.

Транзакция (transaction[227]) — набор операций с базой данных, который представляет собой неделимую логическую единицу. Такой набор операций может быть выполнен либо целиком и успешно (с соблюдением всех правил консистентности базы данных{72} и независимо от параллельно выполняемых транзакций), либо не выполнен вообще (в таком случае ни одна из операций, входящих в данный набор, на должна произвести никаких изменений в базе данных).

Упрощённо: набор операций, который либо целиком и успешно завершается, либо также целиком отменяется в случае ошибки при выполнении любой из его операций.

В приведённом в сноске англоязычном определении[227] также сказано, что транзакция является единицей восстановимости и конкурентности.

Итого, получается, что транзакция:

- всегда или выполняется, или не выполняется целиком;
- используется при восстановлении после различных сбоев и отказов;
- обеспечивает механизм конкурентного доступа к данным.

Продемонстрируем все три свойства графически (см. рисунок 5.5.а). Для простоты возьмём пример, представленный в подавляющем большинстве литературы (но дополним и расширим его): перевод денег между двумя счетами.

Предположим, что два клиента (например, два бухгалтера одной фирмы) почти одновременно переводят средства с некоего счёта («почти», потому что даже в случае совпадения времени до наносекунд всё равно одна из операций будет считаться начатой раньше).

С момента, когда клиент 1 вносит изменения в счёт А, СУБД блокирует действия клиента 2 по изменению этого же счёта (т.к. ещё не ясно, чем закончатся действия клиента 1).

В работе клиента 1 возникает некая ошибка (предположим, что причина находится вне СУБД, т.е. это какие-то проблемы с дисковой подсистемой или нечто подобное).

В такой ситуации завершить перевод денег, выполняемый клиентом 1, уже невозможно (т.к. надо не только уменьшить баланс счёта-источника, но и увеличить баланс счёта-приёмника). Потому операции со счётом В уже не будут выполняться.

Вместо этого СУБД отменит ранее выбранные операции по изменению данных (т.е. по изменению баланса счёта А) и завершит транзакцию (сообщив об ошибке).

---

227 **Transaction** — a unit of recovery and concurrency; loosely, a unit of work. Transactions are all or nothing, in the sense that they either execute in their entirety or have no effect (other than returning a status code or equivalent, perhaps). Transactions are often said to be a unit of integrity (or consistency) also. («The New Relational Database Dictionary», C.J. Date)

Как только транзакция клиента 1 завершена, клиент 2 может продолжить работу со счётом A. В нашем примере в работе клиента 2 никаких ошибок не возникло и его перевод денег со счёта A на счёт C прошёл успешно.

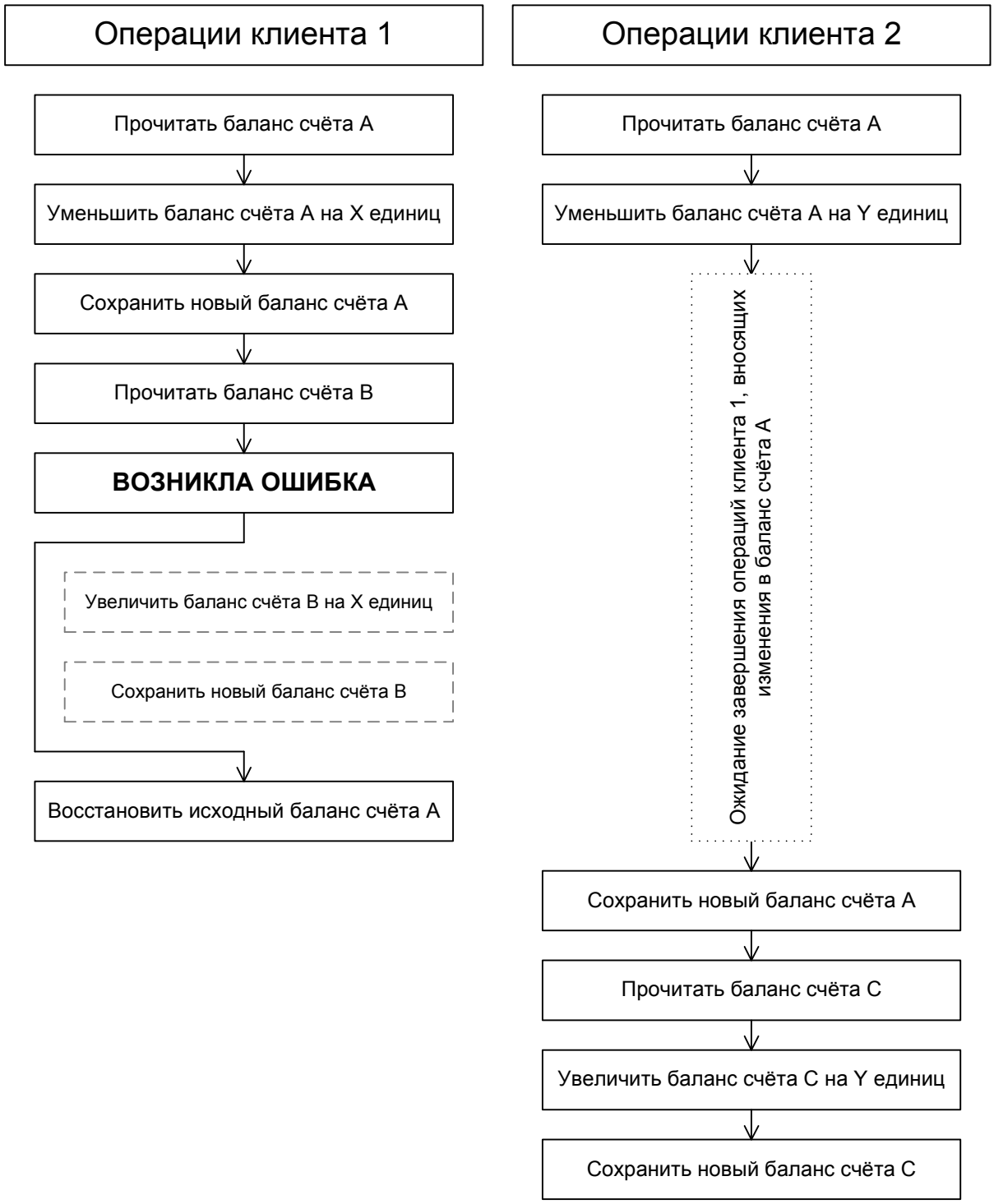


*Рисунок 5.5.a — **Логика работы транзакций***

Итак, ещё раз. Транзакция:

- всегда или выполняется, или не выполняется целиком — при возникновении проблем в транзакции клиента 1 все его изменения были отменены, работа клиента 2 завершилась успешно и все его изменения вступили в силу;
- используется при восстановлении после различных сбоев и отказов — когда в процессе работы клиента 1 возникли проблемы, все его изменения были отменены, а данные были приведены в то состояние, в котором они были до начала операций клиента 1;
- обеспечивает механизм конкурентного доступа к данным — СУБД не позволила двум клиентам одновременно изменять один и тот же счёт.

Из определения транзакции (а до сих пор мы, фактически, развёрнуто рассматривали именно её определение) следуют свойства транзакции[228]. Эти четыре свойства известны под аббревиатурой ACID (по первым буквам англоязычных терминов).

Атомарность (atomicity) означает, что транзакция не может быть «выполнена частично», т.е. что всегда все её операции будут либо выполнены (все и до конца), либо не выполнены (ни одна, т.е. все изменения данных будут отменены). В примере на рисунке 5.5.a транзакция клиента 2 была выполнена полностью до конца, а транзакция клиента 1 была «полностью не выполнена», т.е. произошедшие в процессе её выполнения изменения были аннулированы.

Консистентность (consistency) означает, что успешно завершившаяся транзакция гарантированно сохраняет консистентность базы данных{72}, т.е. фиксирует только допустимые результаты изменения данных (не противоречащие никаким ограничениям, реализованным на уровне СУБД или добавленным отдельно в базу данных в виде проверок{347}, триггеров{350} и т.д.). В отдельных случаях проверки дополнительных условий реализуются непосредственно в коде транзакции.

Важно отметить, что в процессе выполнения транзакции консистентность базы данных{72} может нарушаться — так в примере на рисунке 5.5.a во время выполнения операций обоими клиентами существует момент, когда баланс счёта-источника уже уменьшен, а баланс счёта-приёмника ещё не увеличен, т.е. деньги «пропали в никуда». Однако это внутренне состояние СУБД «не показывает» другим транзакциям, работающим с теми же данными, т.е. извне транзакции нарушения консистентности (почти) никогда не видны. «Почти» потому, что всё же существуют способы на свой страх и риск получить доступ к промежуточным состояниям базы данных — об этом будет упомянуто, когда мы будем рассматривать уровни изолированности транзакций{377}.

Изолированность (isolation) означает, что несколько выполняемых параллельно транзакций не должны влиять на результат выполнения друг друга. Если бы это свойство отсутствовало, мы могли бы получить ситуацию, представленную на рисунке 5.5.b: каждая транзакция в начале читает баланс счёта A, потом изменяет его и в конце проверят, что счёт A изменился на ожидаемую величину. Но поскольку параллельно выполняемая транзакция тоже изменяет этот счёт, естественно, проверка завершится неудачей, т.к. фактический баланс счёта будет отличаться от ожидаемого.

Подробнее об этом свойстве мы поговорим, когда будем рассматривать уже упомянутые уровни изолированности транзакций{377}.

---

228 Особо подчеркнём, что здесь речь идёт о т.н. «теоретических» свойствах транзакций, потому что в реальных СУБД (особенно распределённых) ситуация может быть настолько сложной, что приходится искать различные компромиссы и допускать серию исключений из определяемых этими свойствами правил.

Устойчивость (durability) означает, что СУБД сама решает все «внутренние проблемы» и гарантирует, что после завершения транзакции (как успешного, так и неуспешного) все необходимые изменения были или зафиксированы, или отменены, и что база данных не вернётся в некое «промежуточное состояние». Это свойство, как правило, ассоциируют с устойчивостью к сбоям в аппаратном обеспечении, сетевом взаимодействии и т.д. Даже в таких условиях СУБД всегда «знает», какие операции были выполнены и после возвращения к нормальным условиям работы может удостовериться, что данные не повреждены (или, в самом худшем случае, сообщить о возникших проблемах).

*Рисунок 5.5.b — **Нарушение изолированности транзакций***

Переходим к рассмотрению уже неоднократно упомянутых уровней изолированности транзакций.

Уровень изолированности транзакций (transaction isolation level[229]) — условное значение, показывающее, насколько внутреннее состояние базы данных в момент выполнения транзакции доступно другим, одновременно выполняемым транзакциям.

Упрощённо: насколько параллельно (одновременно) выполняемые транзакции «защищены» друг от друга.

229 **Transaction isolation level** — is a measure of the extent to which transaction isolation succeeds («Transaction Isolation Levels»). [https://docs.microsoft.com/en-us/sql/odbc/reference/develop-app/transaction-isolation-levels?view=sql-server-ver15]

На рисунке 5.5.b представлен пример нарушения изолированности транзакций. Действительно, если бы СУБД позволила двум транзакциям одновременно изменять одни и те же данные, проверка баланса счёта А закончилась бы неудачей (на рисунке 5.5.a показано, что вторая транзакция переводится в режим ожидания, что позволяет избежать этой проблемы).

Но что, если бы нас интересовало не точное изменение баланса счёта А, а просто сам факт того, что баланс изменился? Или если бы мы строили в реальном времени график изменения доступных клиенту средств (где допускали бы некоторые неточности в угоду скорости)? Или если бы возникли какие-то более сложные ситуации, в которых нам нужно было бы обеспечить некое строго заданное поведение СУБД?

Очевидно, что такие вопросы возникали у множества специалистов, работающих с базами данных, и на них уже есть готовые ответ: мы можем управлять уровнем изолированности транзакций, т.е. степенью их взаимного влияния.

Прежде, чем мы рассмотрим сами уровни, необходимо пояснить, какие типичные проблемы могут возникать при одновременном доступе к одним и тем же данным нескольких транзакций.

Потерянное обновление (lost update) — сохраняются только те изменения данных, которые были выполнены позже всего.

Очень упрощённый пример: несколько сотрудников в офисе ругаются по поводу настроек кондиционера, т.е. кто-то его включает, кто-то выключает, кто-то делает теплее, кто-то холоднее — не важно, кто и что сделал ранее, кондиционер всегда настроен так, как его настроили «в самый последний момент».

Графическое пояснение представлено на рисунке 5.5.c.

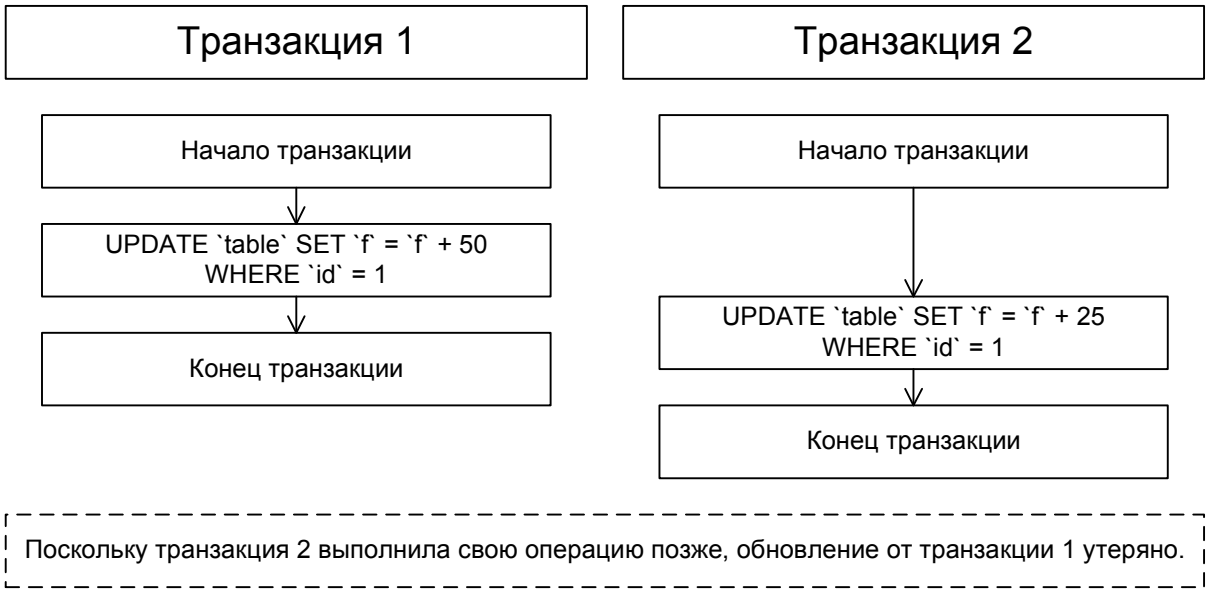


*Рисунок 5.5.c — **Потерянное обновление***

Грязное чтение (dirty read) — становится доступным временное состояние данных, которые в дальнейшем будут удалены или изменены в силу отмены работавшей с ними транзакции.

Очень упрощённый пример: ребёнок подслушал, что родители собираются подарить ему на день рождения велосипед, и радостный побежал рассказывать об этом друзьям; через пять минут родители передумали.

Графическое пояснение представлено на рисунке 5.5.d.

Неповторяющееся чтение (non-repeatable read) — происходит изменение одних и тех же данных за время работы транзакции (т.е. при повторном чтении ранее прочитанных данных получается новый результат).

Очень упрощённый пример: вы решили попить чаю; заглянули в шкафчик и посмотрели, какой там есть чай; пока вы грели чайник, кто-то заменил в шкафчике имевшийся там чай на другой (или вообще на кофе); вы снова открываете шкафчик и сильно удивляетесь, т.к. только что видели там другую картину.

Графическое пояснение представлено на рисунке 5.5.e.

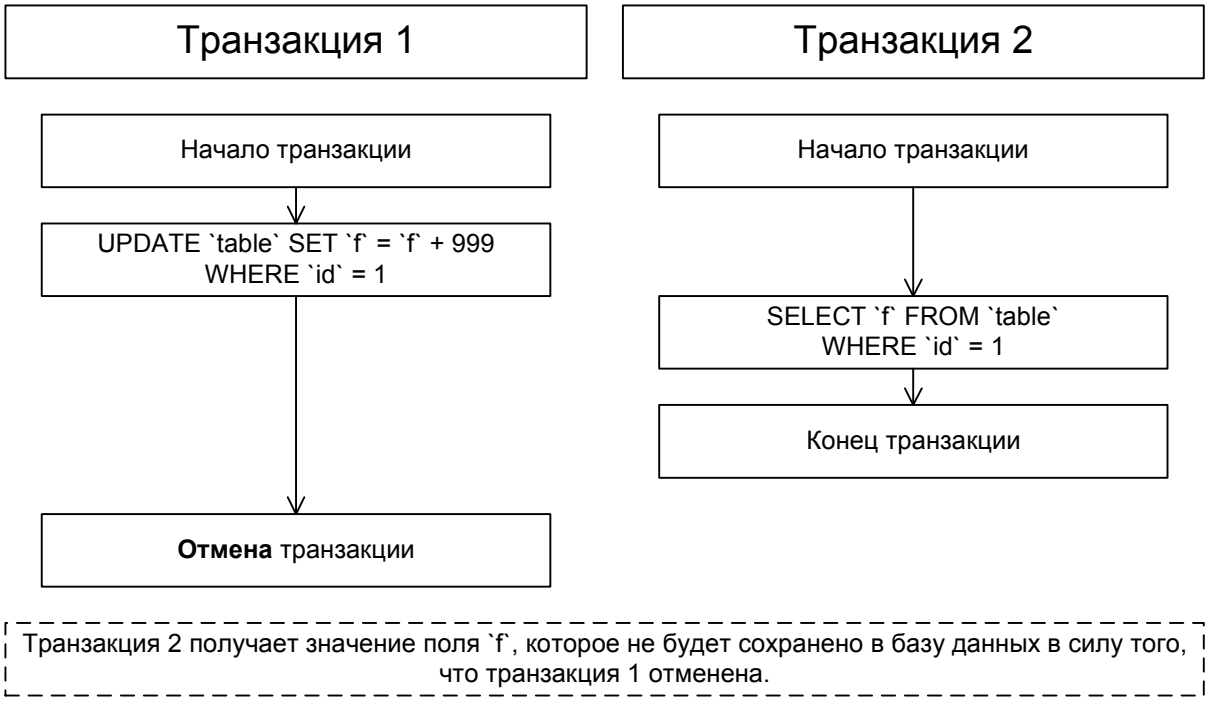


*Рисунок 5.5.d — **Грязное чтение***

*Рисунок 5.5.e — **Неповторяющееся чтение***

Фантомное чтение (phantom reads) — происходит изменение количества строк, подпадающих под выборку (в силу добавления или удаления строк или изменения значений в их полях).

Очень упрощённый пример: вы хотите сфотографировать трёх сидящих на ветке воробьёв; пока вы отвлеклись на настройки фотоаппарата, прилетело ещё два воробья.

Графическое пояснение представлено на рисунке 5.5.f.

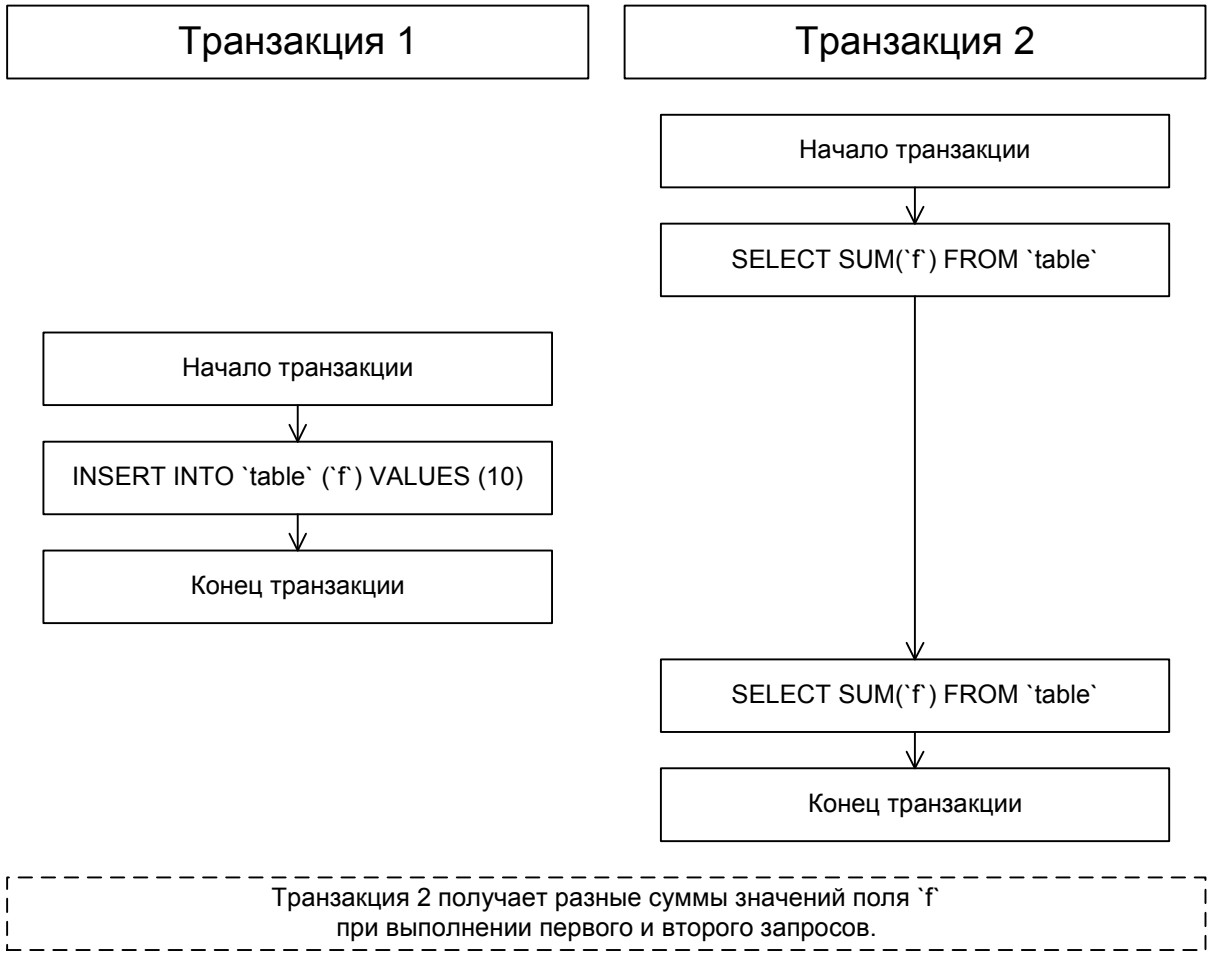


*Рисунок 5.5.f — **Фантомное чтение***

Ключевое отличие неповторяющегося и фантомного чтения состоит в природе их возникновения: при неповторяющемся чтении меняются сами данные, а при повторном чтении меняется *количество* данных (т.е. добавляются или удаляются строки).

Теперь рассмотрим сами уровни изолированности. Когда транзакция выполняется с неким уровнем изолированности, СУБД обеспечивает такой транзакции «защиту» от тех или иных описанных выше проблем. Уровни изолированности представляют собой иерархию, где каждый следующий (более высокий) уровень включает в себя все «защитные механизмы» предыдущих (более низких) уровней.

Иерархия уровней изолированности транзакций представлена на рисунке 5.5.g (поскольку в русскоязычной литературе можно встретить разные варианты перевода этих устоявшихся терминов, на рисунке они осознанно приведены на английском языке).

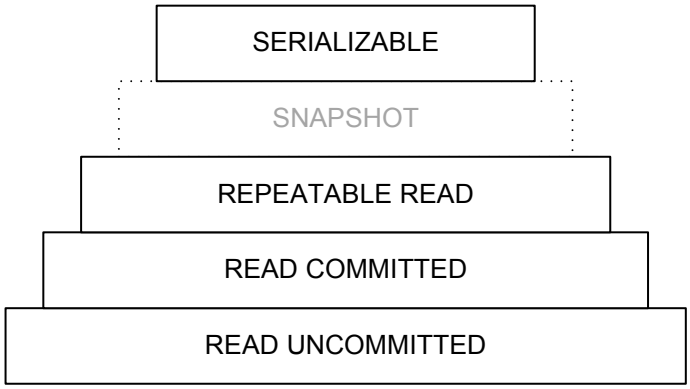


*Рисунок 5.5.g — **Иерархия уровней изолированности транзакций***

Чтение неподтверждённых данных, грязное чтение (read uncommitted, dirty read) — допускает чтение незафиксированных (т.е. до подтверждения или отмены транзакции) изменений — выполненных любой транзакцией (как той, что производит чтение, так и выполняющихся параллельно с ней).

Этот уровень обеспечивает отсутствие потерянных обновлений, т.е. если несколько параллельных транзакций изменяют одни и те же данные, в итоге эти данные будут иметь значение, полученное последовательным применением всех внесённых изменений. Все остальные проблемы (грязное чтение, неповторяющееся чтение, фантомное чтение) продолжают оставаться актуальными.

Физически для защиты от потерянного обновления применяется блокировка изменяемых данных, т.е. изменения одних и тех же данных, выполняемые параллельно, на самом деле выполняются последовательно (выстраиваются в очередь).

Никакие операции чтения на данном уровне изоляции не блокируются.

Чтение подтверждённых данных (read committed) — допускает чтение всех изменений, выполненных самой транзакцией, и только подтверждённых (зафиксированных) изменений, выполненных другими (параллельными) транзакциями.

Этот уровень обеспечивает отсутствие потерянных обновлений и грязного чтения, при этом допускаются неповторяющееся чтение и фантомное чтение.

Физически данный уровень реализуется с использованием блокировки или версионирования данных:

- при блокировке транзакция, изменяющая данные, блокирует чтение этих данных для параллельных транзакций, выполняемых на уровне read committed и более высоких;
- при версионировании СУБД создаёт новую версию (копию) изменяемых данных для той транзакции, которая эти данные изменяет, а всем остальным (параллельным) транзакциям предоставляет доступ к старой (неизменённой) версии.

Оба варианта имеют множество преимуществ и недостатков, а также особенностей реализации, потому подробности можно найти только в документации по конкретной версии вашей конкретной СУБД.

Повторяющееся чтение (repeatable read) — допускает только чтение изменений, выполненных самой транзакцией, а прочитанные ею данные становятся недоступными для изменения параллельным транзакциям.

Этот уровень обеспечивает отсутствие потерянных обновлений, грязного чтения и неповторяющегося чтения, но допускает фантомное чтение.

Физически данный уровень реализуется через блокировку прочитанных данных, что запрещает параллельным транзакциям изменять соответствующие строки таблиц. Но параллельные транзакции могут вставлять новые строки, что может порождать проблему фантомного чтения.

Снимок (snapshot) — является частным (более высоким) случаем повторяющегося чтения, поддерживается не всеми СУБД, и допускает только чтение изменений, выполненных самой транзакцией, а *прочитанные ею данные остаются доступны для изменения параллельным транзакциям* (в этом состоит основное отличие от уровня повторяющегося чтения).

Сериализация (serializable) — допускает только такое выполнение изменений данных, словно все модифицирующие данные транзакции выполняются не параллельно, а последовательно.

Этот уровень обеспечивает отсутствие всех проблем, т.е. потерянных обновлений, грязного чтения, неповторяющегося чтения и фантомного чтения.

Физически это достигается за счёт управления очередью транзакций и сложной системы блокировок. Это самый надёжный в плане точности работы с данными уровень изолированности транзакций, но он же — и самый медленный с точки зрения производительности.

Для простоты и наглядности сведём информацию о том, «что от чего защищает» в единую таблицу.

	Потерянное обновление	Грязное чтение	Неповторяющееся чтение	Фантомное чтение
Чтение неподтверждённых данных	**Защищает**	**Не защищает**	**Не защищает**	**Не защищает**
Чтение подтверждённых данных	**Защищает**	**Защищает**	**Не защищает**	**Не защищает**
Повторяющееся чтение	**Защищает**	**Защищает**	**Защищает**	**Не защищает**
Снимок	**Защищает**	**Защищает**	**Защищает**	**Не защищает**
Сериализация	**Защищает**	**Защищает**	**Защищает**	**Защищает**

Возможно, вас до сих пор волнует вопрос о том, зачем существует столько разных уровней, подходов, решений. Всё это разнообразие позволяет достичь оптимального сочетания необходимой защищённости данных, точности и производительности операций.

В разделе 6.2.2 «Взаимодействие конкурирующих транзакций» книги[230] «Работа с MySQL, MS SQL Server и Oracle в примерах» представлены результаты экспериментов по перекрёстному взаимодействию транзакций во всех возможных уровнях изолированности. Эти результаты показывают, что конкретная реализация транзакционных механизмов в той или иной СУБД может отличаться от теоретических положений.

Задание 5.5.а: составьте список последовательностей операций с базой данных «Банк»{408}, которые следовало бы выполнять внутри одной транзакции. Для каждой получившейся транзакции укажите её минимально допустимый уровень изолированности.

230 «Работа с MySQL, MS SQL Server и Oracle в примерах» (С.С. Куликов) [http://svyatoslav.biz/database_book/]

5.5.2. УПРАВЛЕНИЕ ТРАНЗАКЦИЯМИ

Возможно, после такого достаточно сложного и объёмного теоретического материала вы ожидаете, что управление транзакциями тоже будет выглядеть непросто. И да, и нет.

«Нет» в том плане, что синтаксис управления транзакциями очень прост (сейчас мы его рассмотрим).

«Да» в том плане, что реальная потребность в управлении транзакциями возникает в достаточно сложных ситуациях, а сами механизмы транзакций в различных СУБД имеют множество неочевидных особенностей, и в совокупности это выливается в высокую сложность продумывания, реализации и отладки соответствующих решений.

Но начнём с синтаксиса. Любая транзакция имеет начало и один из двух вариантов завершения — успешный и неуспешный (см. рисунок 5.5.h).

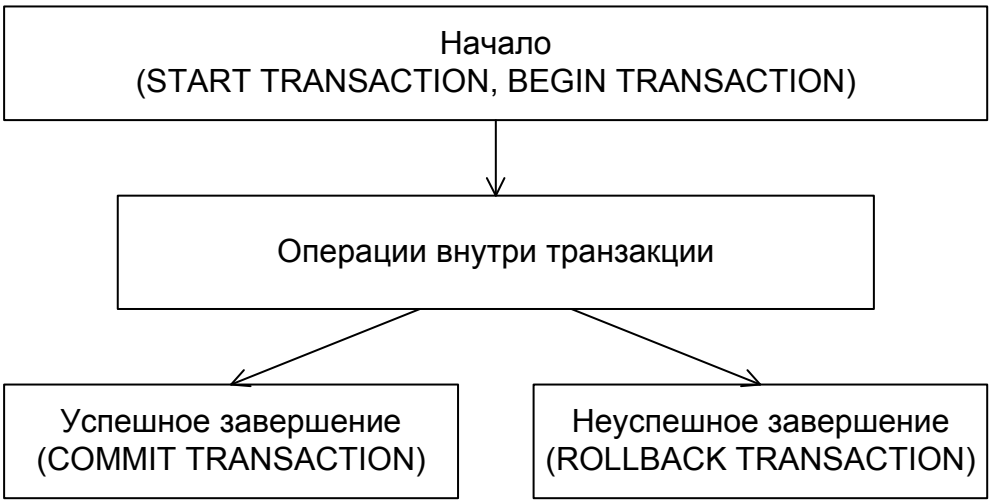


*Рисунок 5.5.h — **Последовательность операций в транзакциях***

Более сложные случаи (вложенные транзакции, возврат к точкам восстановления и т.д.) поддерживаются не всеми СУБД и объективно выходят за рамки данной книги.

Далее для определённости мы будем использовать синтаксис MySQL (тем более, что наша учебная база данных файлообменного сервиса работает именно под управлением этой СУБД):

- начало транзакции обозначается командой `START TRANSACTION`;
- успешное завершение транзакции (фиксация, «коммит») обозначается командой `COMMIT`;
- неуспешное завершение транзакции (отмена, «откат», «роллбэк»[231]) обозначается командой `ROLLBACK`.

[231] Как бы мы ни старались в рамках данной книги придерживаться официальной строгой терминологии, всё же необходимо упомянуть, что в контексте транзакций даже в русскоязычной аудитории почти всегда используются англицизмы наподобие «коммит», «роллбэк» и т.д., т.к. они намного короче, понятнее, универсальнее и проще аналогичных русскоязычных официальных терминов.

Т.е. самый простой случай использования транзакций будет выглядеть так (на примере удаления пользователя с идентификатором 1000):

MySQL **Пример простейшей транзакции**

```
START TRANSACTION;
DELETE FROM `user`
      WHERE `u_id` = 1000;
COMMIT;
```

Однако, можно пойти ещё дальше. Ведь как выполняется «просто запрос» (если мы не обрамляем его командами **`BEGIN TRANSACTION`** / **`COMMIT`**)?

На самом деле — всё происходит точно так же, потому что СУБД реализует т.н. «неявные транзакции» (implicit transactions): фактически, команды **`BEGIN TRANSACTION`** / **`COMMIT`** выполняются без нашего участия.

Этим поведением СУБД даже можно в некоторой степени управлять (в MySQL за автоподтверждение транзакций отвечает параметр **`autocommit`**, и команда **`SET autocommit = 0`** выключает автоподтверждение транзакций, после чего все изменения данных придётся явно фиксировать командой **`COMMIT`**).

Неявные транзакции удобны потому, что очень часто желаемый эффект достигается выполнением всего лишь одного SQL-запроса, и нет необходимости объединять в транзакцию несколько действий: потому логично, что такая «микротранзакция», состоящая из одного запроса, автоматически фиксируется (в случае успешного выполнения запроса) или автоматически отменяется (в случае ошибки выполнения запроса).

Неявные транзакции выполняются на «уровне изолированности по умолчанию» (у каждой СУБД он свой, настраивается в конфигурации СУБД и может быть определён соответствующими командами).

Для определения текущего уровня изолированности (т.е. уровня изолированности по умолчанию, если вы его явно не изменяли) в MySQL используется команда **`SELECT @@TX_ISOLATION`**. Эта СУБД по умолчанию работает на уровне повторяющегося чтения (repeatable read{382}).

Для изменения уровня изолированности транзакций в MySQL используется команда **`SET TRANSACTION ISOLATION LEVEL название_уровня`**.

Перед рассмотрением примера особо подчеркнём, что управление транзакциями (равно как и любой другой инструмент) не является самоцелью. Если при использовании неявных транзакций на уровне изолированности по умолчанию у вас всё работает, вас, заказчика и пользователей всё устраивает, нет никаких проблем — то нет и никакой необходимости усложнять ситуацию.

В качестве примера рассмотрим решение следующих задач:

- быстро построить календарное распределение загрузок файлов;
- гарантированно определить, сколько пользователей находится в каждой из ролей;

Для решения первой задачи важен тот факт, что дата и время загрузки уже загруженного файла (как минимум в теории) не могут изменяться, в то время как другая информация о файлах может изменяться (и факт её изменения приведёт к блокировке соответствующей строки таблицы целиком[232]).

Отсюда мы приходим к выводу, что изменения иных полей (кроме **`f_upload_dt`**) нас при выполнении этой задачи не интересуют, и они не должны никак мешать нам выполнять задачу по построению календарного распределения загрузок.

---

232 Это поведение может отличаться в разных версиях MySQL, при разных настройках метода доступа или разных методах доступа. Данное утверждение приведено как вводное условие для внесения ясности.

Т.е. мы должны читать данные из строк, заблокированных процессом обновления. Т.е. мы должны читать данные из строк, ещё не зафиксированных (uncommitted). Т.е. нам нужен уровень чтения неподтверждённых данных (read uncommitted{382}).

Соответствующий код выглядит следующим образом:

MySQL **Быстрое построение календарного распределения загрузок файлов**

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
START TRANSACTION;
SELECT YEAR(FROM_UNIXTIME(`f_upload_dt`)) AS `Y`,
       MONTH(FROM_UNIXTIME(`f_upload_dt`)) AS `M`,
       DAY(FROM_UNIXTIME(`f_upload_dt`)) AS `D`,
       COUNT(`f_id`) AS `files`
  FROM `file`
 GROUP BY `Y`, `M`, `D`;
COMMIT;
```

В условии второй задачи ключевым словом является «гарантированно», т.е. здесь мы можем немного перестраховаться (да, уровня изолированности по умолчанию repeatable read{382} должно хватить) и выбрать самый защищённый уровень — уровень сериализации (serializable{383}).

Соответствующий код выглядит следующим образом:

MySQL **Гарантированное определение количества пользователей в каждой из ролей**

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
START TRANSACTION;
SELECT `r_name`,
       COUNT(`ur_user`) as `users`
  FROM `role`
  JOIN `m2m_user_role`
    ON `r_id` = `ur_role`;
COMMIT;
```

И ещё один наглядный пример управления (неявными) транзакциями мы уже видели ранее{360} — команда **`SIGNAL SQLSTATE`** в триггерах, приводит к порождению исключительной ситуации и отмене выполняемой транзакции.

Множество более сложных практических примеров приведено в разделе 6 «Использование транзакций» книги233 «Работа с MySQL, MS SQL Server и Oracle в примерах».

Задание 5.5.b: на основе результатов выполнения задания 5.5.a{383} напишите соответствующий SQL-код и проверьте экспериментально свои предположения. Придумайте для каждой последовательности операций из вашего списка не менее трёх «опасных ситуаций», в которых неверный уровень изолированно транзакции может привести к её некорректной работе.

233 «Работа с MySQL, MS SQL Server и Oracle в примерах» (С.С. Куликов) [http://svyatoslav.biz/database_book/]

ОБЕСПЕЧЕНИЕ КАЧЕСТВА БАЗ ДАННЫХ

ОБЕСПЕЧЕНИЕ КАЧЕСТВА БАЗ ДАННЫХ НА СТАДИИ ПРОЕКТИРОВАНИЯ

6.1.1. ОБЩИЕ ПОДХОДЫ К ОБЕСПЕЧЕНИЮ КАЧЕСТВА БАЗ ДАННЫХ НА СТАДИИ ПРОЕКТИРОВАНИЯ

Фактически, весь представленный выше материал данной книги так или иначе затрагивает вопросы качественного проектирования баз данных. Некоторые особо важные вопросы даже рассмотрены явно{12}. Однако можно выделить несколько дополнительных областей, заслуживающих повышенного внимания[234].

Как непосредственно данные, так и структура базы данных для их хранения и обработки должны обеспечивать выполнение следующих принципов.

Полнота. Одной из частых ошибок начинающих разработчиков баз данных является неполный анализ предметной области и, как следствие, отсутствие в базе данных той или иной информации. Например, описывая сотрудника некоей организации указывают лишь ФИО и дату рождения, но забывают про адрес проживания, телефоны (да, как правило, их несколько), адреса электронной почты (тоже несколько), паспортные данные, сведения об образовании и т.д. и т.п.

[234] Подробности можно почитать в статье «What is Data Quality and How Do You Measure It for Best Results?» (Neil Patel) [https://neilpatel.com/blog/data-quality/]

Действительно, не в каждом случае необходимо хранить все эти (и многие другие) данные. Но тем более важным становится тщательное выяснение того, какие данные всё же нужны. И все соответствующие решения должны найти отражение в таблицах базы данных, их полях, связях и т.д.

Уникальность. Данный вопрос уже рассматривался ранее в контексте требований нормализации, где он был представлен как «требование неизбыточности данных{214}».

С момента, как в базе данных в двух и более местах начинают храниться одни и те же данные, возникновение проблем становится исключительно вопросом времени — и эти проблемы *точно* будут.

Если в силу неких непреодолимых (непреодолимых ли?) обстоятельств уникальности хранения части данных не удаётся добиться, следует приложить максимум усилий для того, чтобы встроенными средствами СУБД обеспечить синхронизированность (согласованность) этих данных.

Учёт изменений во времени. Проблемы с соблюдением этого принципа встречаются ещё чаще, чем проблемы с полнотой{387} данных. Ситуация усугубляется тем, что даже заказчик редко задумывается о том, что некоторые данные понадобится обрабатывать с учётом привязки ко времени.

Представим тривиальный пример: две крупных фирмы собираются подписать контракт на поставку широкого спектра продукции; в момент переговоров цена на все позиции устраивала покупающую сторону; в момент подписания контракта цена уже успела измениться, и теперь никто не помнит точных значений цен по сотням позиций продукции на день переговоров.

Если бы база данных учитывала изменение цены во времени, такой проблемы не возникло бы, т.к. цена продукции на день переговоров была бы точно известна.

К этой же проблеме косвенно относится несвоевременное обновление кэширующих таблиц и полей (если таковые присутствуют в базе данных) и в целом выполнение требования актуальности данных{221}.

Корректность (валидность). Выполнение этого принципа часто заставляет искать золотую середину между гибкостью и следованию стандартам и ограничениям.

Должны ли мы, например:

- Жёстко ограничить формат номера телефона?
- Ввести некие правила для «ФИО» и подобных полей?
- Разрешить выбор учебного заведения из списка или допустить собственное написание?

В случае жёстких ограничений мы упрощаем обработку данных, но ставим в тупик пользователей, чьи реальные данные не соответствуют нашим правилам (так, например, что должен указать пользователь в качестве своего учебного заведения, если такового нет в списке, а вписать «руками» свой вариант нельзя?).

В случае увеличения гибкости мы облегчаем жизнь пользователям, но обработка данных иногда превращается в настоящий ад (реальный пример: на одном образовательном ресурсе пользователи смогли «придумать» более 250 вариантов наименования одного и того же университета).

Очевидно, что есть гибридные решения (например, подсказки при вводе и выбор предложенного или возможность добавить свой вариант), но они требуют дополнительных усилий по реализации, имеют свои ограничения и, в конечном итоге всё сводится к особенностям предметной области и решениям заказчика (например, логично разрешить своё написание «ФИО», но нелогично разрешать своё написание номера кредитной карты).

Главное — помнить о том, что такие вопросы возникнут, и прорабатывать их ещё на стадии проектирования базы данных.

Точность. Больше всего этот вопрос касается любых дробных величин. С какой точностью указывать рост или вес пользователя? С какой точностью хранить цену товара? С какой точностью фиксировать время наступления события?

С одной стороны, может показаться, что «чем точнее, тем лучше». Но стоит помнить, что так мы увеличиваем объём хранимых данных и рискуем получить проблемы с точностью вычислений (дробные величины, как правило, хранятся и обрабатываются с приближённой точностью[235]).

С другой стороны, недостаточная точность хранения может не позволить отразить в базе данных реальные свойства некоего человека, объекта, процесса и т.д.

Решение здесь, как и во многих других случаях, лежит в контексте требований предметной области и решений заказчика.

Единообразие представления. Нарушение этого принципа встречается редко и не несёт фатального характера, но всё равно достаточно странно видеть (например), что дата рождения сотрудника хранится с точностью до дня, а дата рождения ребёнка сотрудника хранится с точностью до секунды. Или что рабочий телефон сотрудника хранится с кодом страны и оператора, а личный — без этих кодов (или что эти номера хранятся просто в разных форматах).

Такое бессмысленное разнообразие сильно нарушает требование удобства использования{12} базы данных.

В завершении этой главы приведём небольшую градацию качества[236] баз данных, которая позволит вам бегло оценить, насколько ваша база данных хороша (или требует улучшений).

Уровень	Описание	Примеры проблем
5	Не обнаружено сколь бы то ни было серьёзных проблем. База данных логична, последовательна, допускает простую поддержку и доработку.	-
4	Возникают проблемы, не влияющие на работающие с базой данных приложения. Такие проблемы достаточно легко обнаружить и исправить.	Типы данных не унифицированы, обязательные поля не отмечены как `NOT NULL`, не проставлены уникальные индексы, имена полей не всегда очевидны.
3	С базой данных есть серьёзные проблемы, очевидно влияющие на работу пользователей (сбои, низкая производительность и т.д.)	Не проставлены первичные ключи, не проставлены индексы (в т.ч. на внешних ключах), типы данных требуют конвертации во многих запросах, нарушена часть рассмотренных в начале данной главы принципов.
2	С базой данных есть критические проблемы, периодически приводящие к неработоспособности работающих с ней приложений.	Нарушено большинство рассмотренных в начале данной главы принципов, присутствуют аномалии работы с данными{161}.
1	База данных практически неработоспособна.	Нарушены все рассмотренные в начале данной главы принципы, нарушены фундаментальные требования к базам данных{12}, присутствуют многочисленные аномалии работы с данными{161}.

К сожалению, нередко получается так, что разработчикам базы данных результат их труда представляется как относящийся к уровню 5, но в реальности этот уровень может оказаться намного ниже.

Именно поэтому процесс проектирования баз данных является такой длительной и нетривиальной задачей, а его результат требует такой тщательной проверки.

---

235 Этому вопросу даже посвящён отдельный сайт, на котором приведено много примеров и пояснений. См. https://0.30000000000000004.com.

236 Подробности можно прочитать в статье «Grading Database Quality» (Michael Blaha) [https://www.dataversity.net/grading-database-quality-part-1-database-designs/]

Подробнее об обеспечении качества «в принципе» можно почитать в книге «Тестирование программного обеспечения. Базовый курс.[237]», а непосредственно о качестве данных и баз данных — в книге «Information and Database Quality[238]».

Задание 6.1.a: нарушение каких из рассмотренных в данной главе принципов вы можете обнаружить в базе данных «Банк»{408}? Составьте список с указанием сути нарушения и рекомендациями по исправлению ситуации.

Задание 6.1.b: нарушение каких из рассмотренных в данной главе принципов вы можете обнаружить в базе данных файлообменного сервиса{284}, {302}, {314}? Составьте список с указанием сути нарушения и рекомендациями по исправлению ситуации.

237 «Тестирование программного обеспечения. Базовый курс.» (С.С. Куликов) [http://svyatoslav.biz/software_testing_book/]

238 «Information and Database Quality» (Mario G. Piattini, Coral Calero, Marcela Genero).

6.1.2. ТЕХНИКИ И ИНСТРУМЕНТЫ ОБЕСПЕЧЕНИЯ КАЧЕСТВА БАЗ ДАННЫХ НА СТАДИИ ПРОЕКТИРОВАНИЯ

Как мы помним, процесс проектирования баз данных может быть нисходящим{11} и восходящим{12}. И во втором случае крайне сложно сконцентрироваться исключительно на проектировании, не затронув хотя бы частично вопрос эксплуатации базы данных.

Но такое строгое разделение и не требуется. Здесь мы всего лишь больше внимания уделим вопросам качества на стадии проектирования, а про эксплуатацию подробнее поговорим в следующем разделе{394}.

К сожалению, не существует некоего «магического» инструмента, который позволил бы сразу и однозначно показать ошибки проектирования. И это — фундаментальная проблема, т.к. даже самый совершенный инструмент не обладает полной информацией о предметной области, пожеланиях заказчика и потребностях пользователей.

Т.е. самым эффективным «инструментом» в данном случае является специалист, занимающийся проектированием базы данных.

Поскольку невозможно построить модель сколь бы то ни было сложной базы данных без специальных средств, примеры соответствующих решений были рассмотрены ранее для проектирования на инфологическом{286}, даталогическом{304} и физическом{319} уровнях.

Но эти решения позволяют лишь упростить процесс сбора, анализа и отображения информации, необходимой для успешного проектирования качественной базы данных.

Основная же часть нагрузки приходится на соответствующие действия (техники, подходы), наиболее актуальные из которых мы сейчас рассмотрим[239].

Сбор, анализ и обсуждение информации. Существует большое количество источников информации при проектировании баз данных (например, заказчик, пользователи, разработчики работающих с базой данных приложений, объективно существующие ограничения предметной области и т.д.) — каждый такой источник должен быть выявлен, а поступающая от него информация получена, обсуждена с остальными заинтересованными сторонами и отражена в модели базы данных.

Если мнение хотя бы одной заинтересованной стороны было проигнорировано, можно смело утверждать, что база данных не в полной мере будет соответствовать потребностям тех людей, для которых она создавалась.

Выбор технических решений. Как правило, проблемы здесь возникают редко, но носят фатальный характер. Выбор вида СУБД, конкретной версии и реализации этой СУБД, сопутствующей инфраструктуры — всё это может (и часто должно) накладывать отпечаток на модель создаваемой базы данных. Здесь важно получить обоснованные решения и получить их заблаговременно, чтобы не было необходимости переделывать часть уже выполненной работы.

Контроль и самоконтроль. Даже в идеальных условиях, почти недостижимых в реальной жизни (когда вся информация есть, все заинтересованные стороны со всем согласны, ничего не забыто и т.д.) всё равно очень высока вероятность что-то перепутать, забыть, неправильно учесть и неверно отразить в модели базы данных.

---

239 Здесь осознанно не рассматриваются вопросы проектирования баз данных корпоративного уровня, т.к. они затрагивают процессы управления проектом, обучения персонала, управления качеством и многие другие вопросы, выходящие за рамки книги для начинающих специалистов.

Потому так важно многократно перепроверять результаты своей работы, получать обратную связь от коллег-специалистов и от всех заинтересованных сторон. Также здесь очень хорошо помогает уже упомянутое ранее обсуждение информации, а также — эксперименты.

Эксперименты. Начиная с даталогического уровня проектирования{302}, уже можно периодически экспортировать модель базы данных в СУБД, наполнять её тестовыми данными и исследовать поведение. Такой подход позволяет не только увидеть проблемы с производительностью и иные сложнопредсказуемые сложности, но также покажет ряд досадных грубых ошибок (как правило, появившихся в силу невнимательности и легко отслеживаемых средствами самой СУБД — от которой вы будете получать вполне конкретные сообщения о конкретных ошибках).

«А что, если?» vs «Ура, работает!» Типичной ошибкой начинающих специалистов является прекращение анализа проблемы после того, как найдено первое же работающее решение. Но, во-первых, это решение может не быть самым лучшим (и дальнейший анализ позволил бы найти лучший вариант), а во-вторых, такое первое попавшееся решение может покрывать лишь частные случаи, и в других ситуациях (с другими данными, другими условиями выполнения, другим способом взаимодействия с базой данных и т.д.) приводить к возникновению ошибок.

Потому поиск нескольких альтернативных решений, выбор наилучшего из них и анализ выбранного решения в различных ситуациях ощутимо повышает качество разрабатываемой базы данных.

Тщательное документирование. Предположительно, над более-менее сложной базой данных будет работать несколько человек на протяжении длительного времени. Как не забыть то, что сам придумал полгода назад? Как избавить коллег от необходимости постоянно удивляться, искать автора того или иного решения и расспрашивать его о подробностях? Ответ прост (и он один, другого нет) — документация.

От комментариев к таблицам и их полям, до полноценных графических схем и сопровождающих их подробных текстовых описаний — всё это становится тем более необходимым, чем более масштабную базу данных приходится проектировать.

Расширение собственного кругозора. Эта техника не относится напрямую к базам данных (она в равной степени помогает в любой деятельности), но тем не менее: чем больше вы знаете о предметной области, чем больше разбираетесь в базах данных и сопутствующих технологиях (операционных системах, компьютерных сетях, языках программирования и т.д.), тем легче вы будете замечать потенциально проблемные места, тем проще вам будет задать правильные вопросы и, получив на них ответы, повысить качество создаваемой базы данных.

Конечно, хочется верить, что вам повезёт работать в тщательно организованных проектах с развитой системой обеспечения качества, а своим опытом с вами будут делиться настоящие профессионалы — в такой идеальной обстановке многие из приведённых здесь рекомендаций будут усвоены автоматически.

Но даже если вы в одиночку создаёте свою первую базу данных, представленные выше идеи уже применимы, и уже помогут вам повысить качество проектируемой базы данных.

Задание 6.1.c: какие из представленных в данной главе техник и подходов вызывали у вас наибольшую сложность при работе с базой данных «Банк»{408} и базой данных файлообменного сервиса{284}, {302}, {314}? Составьте список и по каждому пункту напишите, какой информации (или какого навыка) вам не хватало.[240]

[240] Если вы только начинаете заниматься базами данных, сохраните этот список на будущее, и вернитесь к нему через некоторое время. Вы будете приятно удивлены тем, как изменилась ситуация.

6.2. ОБЕСПЕЧЕНИЕ КАЧЕСТВА БАЗ ДАННЫХ НА СТАДИИ ЭКСПЛУАТАЦИИ

6.2.1. ОБЩИЕ ПОДХОДЫ К ОБЕСПЕЧЕНИЮ КАЧЕСТВА БАЗ ДАННЫХ НА СТАДИИ ЭКСПЛУАТАЦИИ

Если в предыдущей главе эксперименты{392} были упомянуты лишь как одна из техник, то на стадии эксплуатации баз данных экспериментальные проверки и наблюдения (мониторинг) становятся основным источником информации для оценки и повышения качества.

Также, поскольку эксплуатация баз данных по определению является предельно практико-ориентированной, в данном случае достаточно сложно отделить общие идеи от техник и инструментов.

Но в общем случае к таким идеям можно отнести следующее.

Обеспечение качества — непрерывный процесс. Конечно, мы постоянно думаем о качестве в процессе проектирования базы данных и особенно тщательно всё проверяем перед полноценным запуском всей системы в эксплуатацию. Но на этом работа не заканчивается.

Во-первых, большинство проектов являются «долгоиграющими», а потому у заказчика будут появляться новые идеи, придётся вносить правки в базу данных и снова повторять многие этапы тестирования.

Во-вторых, с ходом времени ситуация, в которой работает база данных, может измениться. Обновляется СУБД и операционная система, меняется сетевая инфраструктура, изменяется количество работающих с базой данных пользователей, увеличивается объём хранимых и обрабатываемых данных, происходят иные явные и неявные изменения. И всё это способно повлиять на работу базы данных (увы, как правило, негативно).

Потому здесь невозможен вариант «сделать и забыть». С базой данных придётся работать непрерывно до того дня, пока она не будет выведена из эксплуатации.

Чем раньше начинаются эксперименты — тем лучше. Эта идея проистекает из фундаментального принципа тестирования: чем раньше обнаружена проблема, тем проще, быстрее и дешевле её исправить.

«Эксплуатация» (в кавычках, т.к. это, пока, внутренняя эксплуатация) базы данных начинается с первого же эксперимента по импорту модели базы данных в СУБД. И ничто не мешает уже к этому моменту подготовить наборы данных и специальные инструменты и провести серию испытаний, чтобы обнаружить и сразу же исправить широкий спектр проблем (которые в т.ч. могут радикально повлиять на весь процесс проектирования).

К тому же вариант «пусть оно поработает, а потом, когда сломается, мы починим» в большинстве случаев категорически недопустим (ни заказчику, ни пользователям не захочется ждать, пока будет восстановлена работоспособность некоего критического сервиса).

Эксперименты стоит приближать к реальности. Здесь хочется задать несколько риторических вопросов тем людям, которые нарушают это правило. Вы действительно думаете, что в вашей системе будет всего пять пользователей? И на вашем сайте действительно все 100 миллионов новостей будут опубликованы одним втором в одно и то же время? И у всех товаров будет одинаковая цена? И на ваш сайт никогда не зайдёт одновременно больше 2-3 посетителей?

Чем больше эксперимент похож на реальную жизнь, тем больше полезной информации он предоставляет. Потому данные (как по количеству, так и по сути), нагрузка, конфигурация инфраструктуры и т.д. — всё должно быть настолько похоже на реальные условия эксплуатации, насколько это возможно.

Есть одно исключение: если эксплуатация системы подразумевает использование очень масштабной и дорогой инфраструктуры (сотни серверов в десятках дата-центров), такие масштабные исследования требуют отдельного подхода. Но даже в подобной ситуации ничто не мешает использовать для тестирования один сервер на 100% соответствующий тем, на которых в дальнейшем будет работать база данных.

И также ничто не помешает подготовить тестовые данные в некотором масштабе (например, 1% от запланированного объёма, если он измеряется петабайтами) и провести эксперименты с соответствующими поправочными коэффициентами.

Наблюдение (мониторинг) дополняет эксперименты. Иногда достаточно просто смотреть, как развиваются события. Современные СУБД (и дополняющие их инструменты) позволяют собирать и анализировать практически в реальном времени данные по тысячам параметров, а также строить на основе таких собранных данных прогнозы и предупреждать о потенциальных проблемах.

Глупо было бы игнорировать такие возможности, часто доступные бесплатно (в комплекте с лицензией на СУБД).

К тому же некоторые эксперименты крайне сложно придумать и полноценно выполнить, а мониторинг может показать (заблаговременно) развитие даже самых маловероятных и неожиданных неприятных ситуаций.

Автоматизация позволяет ощутимо сократить трудозатраты. Только что упомянутый мониторинг вовсе невозможен без автоматизации — только специализированные инструменты способны собрать и проанализировать такой огромный набор данных с необходимой скоростью.

Но даже если мы говорим о классических экспериментах, они включают в себя множество операций, автоматизация которых очень выгодна:

- подготовка исходного состояния системы (возможно, с развёртыванием виртуальных машин и т.д.);
- генерация тестовых данных;
- наполнение базы данных тестовыми данными;
- выполнение запросов с заданной интенсивностью;
- мониторинг работы базы данных и СУБД в процессе эксперимента;
- сбор информации об ошибочных ситуациях;
- генерация сводных отчётов.

Всё это и многое другое можно и нужно автоматизировать. Такой подход позволит с минимумом трудозатрат многократно повторять трудоёмкие эксперименты.

А некоторые эксперименты без автоматизации невозможны в принципе (попробуйте вручную создать информацию о миллиарде товаров, например).

Стоит помнить об инфраструктуре. Иногда проблема (и её решение) может лежать вне базы данных или СУБД. Иногда сложности возникают в операционной системе, или в сетевой инфраструктуре, или в неких сопутствующих приложениях, или в аппаратном обеспечении.

Упомянутый выше мониторинг позволяет отследить даже такие сложности.

А соответствующие эксперименты позволяют проверить поведение базы данных в подобных ситуациях.

Иногда можно услышать возражения, что инфраструктура — это задача DevOps[241]-инженеров, и не нужно пытаться решить её проблемы средствами баз данных. Отчасти такое мнение оправдано, но вам как разработчику базы данных не станет легче от того, что после (например) перезагрузки одного из серверов не восстановится процесс репликации или возникнут иные подобные проблемы.

Да, мы надеемся, что инфраструктура будет работать хорошо, но мы должны быть готовы и к таким ситуациям, когда инфраструктура нас подводит.

Не сломаете вы, сломают другие. И это относится не только к вопросам безопасности (хоть и ими ни в коем случае не стоит пренебрегать). Это относится к любой потенциальной проблеме, особенно когда нам кажется, что «ну уж такого точно не произойдёт».

Базы данных работают в очень сложной аппаратно-программной среде, включающей в себя сотни и тысячи компонентов. В любом из этих компонентов могут быть проблемы. Любой из этих компонентов может быть неправильно сконфигурирован. Любую часть этой сложной системы могут атаковать злоумышленники.

Намного выгоднее предусмотреть соответствующие проблемы и проверить, что наша база данных к ним устойчива, чем в критической ситуации судорожно искать решение и справляться с очень неприятными последствиями.

В завершение этой главы отметим, что производители СУБД сами уделяют много внимания вопросам качества баз данных на стадии эксплуатации, а потому вы всегда можете найти множество статей и видеороликов с конкретными рекомендациями по выбранной вами СУБД.

Задание 6.2.a: какие из представленных в данной главе идеи вы использовали при работе с базой данных «Банк»{408} и базой данных файлообменного сервиса{284}, {302}, {314}? Составьте список и по каждому пункту напишите не менее трёх примеров технических решений, появившихся в результате применения соответствующей идеи.

241 **DevOps** — a set of practices that combines software development (Dev) and IT operations (Ops). It aims to shorten the systems development life cycle and provide continuous delivery with high software quality. («Wikipedia») [https://en.wikipedia.org/wiki/DevOps]

6.2.2. ТЕХНИКИ И ИНСТРУМЕНТЫ ОБЕСПЕЧЕНИЯ КАЧЕСТВА БАЗ ДАННЫХ НА СТАДИИ ЭКСПЛУАТАЦИИ

Оперируя терминологией из области тестирования программного обеспечения, можно сказать, что на стадии проектирования баз данных акцент смещён в сторону обеспечения качества, а на стадии эксплуатации — в сторону контроля качества (эти и многие другие термины подробно пояснены в соответствующей книге[242], посвящённой тестированию программного обеспечения).

Иными словами — здесь необходимо применять различные техники тестирования, направленные на оценку соответствия базы данных и процесса её работы заявленным критериям качества, а также на поиск существующих и потенциальных проблем и ошибок.

И подавляющее большинство таких техник использует в качестве базового инструмента проведение экспериментов и наблюдение (мониторинг).

В общем и целом, тестирование базы данных можно разделить на три условных подтипа:

- структурное тестирование, направленное на проверку всей схемы базы данных, отдельных таблиц{24} и их полей{23}, представлений{340}, хранимых подпрограмм{363} и т.д.;
- функциональное тестирование, направленное на проверку работы базы данных с точки зрения выполняемых с нею операций;
- нефункциональное тестирование, направленное на проверку производительности, безопасности и иных подобных показателей качества.

В процессе тестирования базы данных, как правило, необходимо выполнить несколько последовательных действий:

- Настроить среду выполнения теста. В зависимости от различных условий эта задача может подразумевать очистку базы данных от имеющихся там данных, наполнение тестовыми данными, настройку окружения и иные процедуры.
- Выполнить тест. Здесь варианты реализации могут варьироваться от выполнения вручную нескольких запросов до запуска сложного механизма тестирования с применением специальных средств автоматизации.
- Проанализировать результаты теста. В идеальном случае они будут соответствовать нашим ожиданиям, т.е. «всё работает так, как надо». Но гораздо чаще мы заметим различные отклонения, ошибки, проблемы. Необходимо определить их причины.
- Сформировать отчёт о результатах тестирования. Эта процедура относится не столько к тестированию баз данных, сколько к «тестированию в принципе» (и подробно описана в соответствующей книге[242]).

---

[242] См. подробнее в книге «Тестирование программного обеспечения. Базовый курс.» (С.С. Куликов) [http://svyatoslav.biz/software_testing_book/]

Если по каким-то причинам тестирование необходимо провести на реальной работающей базе данных, то стоит запомнить золотое правило: этого никогда нельзя делать. Без исключения.

За реальной работающей базой данных можно только наблюдать (проводить мониторинг). Всё тестирование необходимо выполнять только на её копии (в идеале такая копия должна быть не «копией данных», а копией всей среды исполнения — вплоть до операционной системы, что очень легко достигается с использованием виртуальных машин).

Очевидно, что цели, задачи, способы и инструменты тестирования стоит подбирать в конкретной ситуации, исходя из проектной обстановки.

Но для лучшего понимания базовых принципов рассмотрим несколько универсальных идей, которые могут стать отправной точкой, чтобы протестировать…

- Таблицы и их поля:
  - Все таблицы в реальной базе данных присутствуют, имеют ожидаемые имена, структуру, типы полей.
  - У всех таблиц корректно определены первичные ключи (особенно — составные).
  - У всех полей корректно установлены свойства `NULL` / `NOT NULL`, значения по умолчанию, уникальные индексы, свойства «автоинкрементируемости».
  - Присутствуют все необходимые связи, на внешних ключах построены индексы, типы первичных и внешних ключей полностью совпадают.
  - На всех необходимых полях установлены проверки{347}.
- Хранимые подпрограммы:
  - Все хранимые подпрограммы присутствуют, имеют ожидаемые имена, имеют ожидаемые наборы входных и выходных параметров (включая типы данных).
  - Возможна ли ситуация, когда подпрограмма вернёт больше или меньше данных, чем ожидаемся (например, пустой набор записей вместо нескольких записей, или несколько записей вместо одной, или `NULL`-значение вместо «настоящего» значения)?
  - Возможна ли передача в хранимую подпрограмму таких параметров, на которые она не рассчитана, и какова её реакция? Например, передано слишком большое число, или слишком длинная строка, или строка нулевой длины (хотя ожидалось, что в строке всегда будут символы)?
  - Генерирует ли хранимая подпрограмма корректное сообщение об ошибочной ситуации (как непосредственно текст сообщения, так и тип порождённого исключения и т.д.)?
- Триггеры:
  - Все необходимые триггеры присутствуют в базе данных, у них корректные имена, они созданы на правильных таблицах и реагируют на правильные события.
  - Возможна ли ситуация, в которой поведение триггера будет отличаться от ожидаемого?
  - Генерирует ли триггер корректное сообщение об ошибочной ситуации (как непосредственно текст сообщения, так и тип порождённого исключения и т.д.)?
- Настройки базы данных и СУБД:
  - Корректно ли выполняются все операции по импорту модели базы данных в СУБД, наполнению её тестовыми данными, конфигурированию как самой базы данных, так и СУБД?
  - Успешно ли взаимодействуют с базой данных работающие с ней приложения? Корректно ли проходит тестирование таких приложений, затрагивающее взаимодействие с базой данных?
- Производительность:
  - Соответствует ли производительность базы данных ожидаемому уровню при планируемой нагрузке (с учётом реального набора данных, количества пользователей, сути и интенсивности выполняемых ими операций)?

- Сообщает ли СУБД своими встроенными механизмами о потенциальных проблемах (нехватке оперативной памяти или процессорной мощности, наличии «долгоиграющих» запросов, периодических взаимных блокировках транзакций и иных узких местах)?
- Существуют ли проблемы с производительностью работающих с базой данных приложений?
- Насколько быстро работают наиболее часто выполняемые запросы (именно они оказывают наиболее ощутимое влияние на производительность?

Ещё одним эффективным способом понимания тестирования баз данных является проведение параллелей между проверками в работающих с базой данных приложениях и действиями непосредственно с базой данных. Представим это в виде таблицы[243]:

Тестирование приложений	Тестирование баз данных
Подразумевает проверку работоспособности приложений и их компонентов и процессов, таких как формы, меню, отчёты, навигация между страницами и т.д.	Подразумевает проверку невидимых пользователю компонент и процессов, таких как таблицы, триггеры, передача данных между СУБД и приложением.
Необходимо хорошее понимание бизнес-требований и предметной области, функционала приложения и типичных пользовательских сценариев.	Необходимо не только хорошее понимание бизнес-требований и предметной области, функционала приложения и типичных пользовательских сценариев, но и принципов построения и работы баз данных, языка SQL, принципов работы серверной инфраструктуры.
Взаимодействие с приложением выполняется вручную или с помощью средств автоматизации тестирования.	Взаимодействие с СУБД выполняется, как правило, с использованием средств автоматизации и подразумевает намного более глубокое исследование (на различных наборах данных, в различных типичных и нетипичных ситуациях, в которых базе данных придётся функционировать).

Инструменты, которые понадобятся на данной стадии обеспечения качества баз данных, также можно разделить на несколько категорий, в каждой из которых будут приведены примеры конкретных программных средств[244]:

- Инструменты разбора, стандартизации и генерации данных позволяют как подготовить наборы тестовых данных в едином унифицированном формате, так и помогают в решении аналогичных проблем стандартизации и унификации в уже работающих базах данных (DTM Data Generator[245], Turbo Data[246]).
- Инструменты очистки и слияния данных позволяют обнаружить некорректные и дублирующиеся данные, а также выполнить необходимые операции по исправлению ситуации (OpenRefine[247], WinPure[248]).

---

243 Оригинальная идея взята из курса «Database Testing» (там же можно найти много дополнительных подробностей и полезных идей). [https://www.tutorialspoint.com/database_testing/database_testing_overview.htm]

244 Стоит помнить, что индустрия программного обеспечения развивается очень стремительно, потому отдельные примеры в данном списке могут устаревать буквально за несколько недель.

245 «DTM Data Generator» [http://www.sqledit.com/dg/]

246 «Turbo Data» [http://www.turbodata.ca]

247 «OpenRefine» [https://openrefine.org]

248 «WinPure» [https://winpure.com]

- Инструменты профилирования позволяют проводить мониторинг протекающих в СУБД и серверном окружении процессов, оценивать их качество по заданным параметрам и сообщать об имеющихся или потенциальных проблемах (Neor Profile SQL[249], SQL Server Profiler[250]).
- Инструменты нагрузочного тестирования позволяют оценить производительность базы данных (Benchmark Factory[251], Database Benchmark[252]).
- Инструменты низкоуровневого (в т.ч. модульного) тестирования позволяют автоматизировать широкой спектр проверок базы данных (SQL Test[253], tSQLt[254]).

Поскольку дальнейшее углубление в тему данной главы уже выходит за рамки работы с базами данных и переходит в более широкую область тестирования программного обеспечения, предлагаем продолжить погружение в подробности, представленные в книге «Тестирование программного обеспечения. Базовый курс.[255]»

Задание 6.2.b: примените представленные в данной главе техники для анализа базы данных «Банк»{408} и базы данных файлообменного сервиса{284}, {302}, {314}. Составьте список проблем, обнаруженных в процессе такого анализа. По каждой проблеме напишите предложение по её устранению.

249 «Neor Profile SQL» [http://www.profilesql.com]

250 «SQL Server Profiler» [https://docs.microsoft.com/en-us/sql/tools/sql-server-profiler/sql-server-profiler]

251 «Benchmark Factory» [https://www.quest.com/products/benchmark-factory/]

252 «Database Benchmark» [http://stssoft.com/products/database-benchmark/]

253 «SQL Test» [https://www.red-gate.com/products/sql-development/sql-test/]

254 «tSQLt» [https://tsqlt.org]

255 «Тестирование программного обеспечения. Базовый курс.» (С.С. Куликов) [http://svyatoslav.biz/software_testing_book/]

6.3. ДОПОЛНИТЕЛЬНЫЕ ВОПРОСЫ ОБЕСПЕЧЕНИЯ КАЧЕСТВА ДАННЫХ И КАЧЕСТВА БАЗ ДАННЫХ

6.3.1. ТИПИЧНЫЕ ЗАБЛУЖДЕНИЯ ОТНОСИТЕЛЬНО ДАННЫХ

Поскольку базы данных (по определению) занимаются хранением и обработкой данных, важно понимать:

- какие форматы представления данных использовать;
- какие значения могут принимать данные;
- как верно обрабатывать данные;
- какие ограничения на форматы, значения и правила обработки данных следует (или, наоборот, не следует) реализовывать.

В данном контексте стоит отметить, что начинающие разработчики баз данных часто подвержены различным заблуждениям, многие из которых на первый взгляд кажутся совершенно логичными.

Так, например, логично же, что у одного человека будет только один паспорт? Отчасти — да, но — не всегда (и мы такой случай уже рассматривали{167}).

В качестве наглядной иллюстрации мы рассмотрим типичные заблуждения относительно времени[256], имён[257], телефонных номеров[258] и адресов[259].

Да, во многих случаях представленные здесь идеи будут неактуальными для вашей базы данных. И всё же стоит о них помнить (и задавать соответствующие уточняющие вопросы заказчикам).

Информация о дате и времени так или иначе присутствует практически в любой базе данных. И сами понятия «даты» и «времени» настолько плотно присутствуют в нашей жизни, что, казало бы, мы знаем о них всё, и всё всегда понятно.

Но следующие привычные утверждения относительно даты и времени — ложны:

- в сутках всегда строго 24 часа (нет, если был переход на зимнее/летнее время);
- в месяце всегда 30 или 31 день (нет, забыли про февраль);
- в году всегда 365 дней (нет, в високосном году 366 дней);
- в феврале всегда 28 дней (нет, в високосном году в феврале 29 дней);

---

256 «Заблуждения программистов относительно времени» [https://habr.com/ru/post/146109/], «Заблуждения большинства программистов относительно “времени”» [https://habr.com/ru/post/313274/], «Заблуждения программистов о Unix-времени» [https://habr.com/ru/post/452584/]

257 «Заблуждения программистов об именах» [https://habr.com/ru/post/146901/], «Заблуждения программистов об именах — с примерами» [https://habr.com/ru/post/431866/]

258 «Заблуждения программистов о телефонных номерах» [https://habr.com/ru/post/279751/]

259 «У семи программистов адрес без дома» [https://habr.com/ru/company/hflabs/blog/260601/]

- любой 24-часовой период начинается и заканчивается в тот же день, неделю, месяц (нет, если начало периода не совпадает с началом дня);
- неделя всегда начинается и заканчивается в тот же месяц (нет, просто не надо путать абстрактное понятие «месяц» и реальный календарь);
- неделя и месяц всегда начинаются и заканчиваются в тот же год (нет, по аналогии с предыдущим пунктом);
- сервер, на котором работает база данных, всегда будет в часовом поясе UTC+0 (гринвичское время) или в вашем локальном часовом поясе (нет, вы никак не можете предугадать настройки на реальном сервере, на котором будет работать ваша база данных);
- часовой пояс сервера, на котором работает база данных, не будет изменяться (нет, аналогично предыдущему пункту — в любой момент такие настройки могут быть изменены);
- системные часы сервера, на котором работает база данных, всегда идут точно (нет, аналогично двум предыдущим пунктам — часы на сервере могут быть вообще не синхронизированы с источником точного времени, а также ощутимо «отставать» и «спешить» (вплоть до «на несколько минут в сутки»));
- часы на сервере, на котором работает база данных, и на компьютере клиента всегда показывают одинаковое время (нет, т.к. сервер и клиент могут быть в разных часовых поясах, а также и на сервере, и на клиенте часы могут идти неточно);
- часы на сервере, на котором работает база данных, никогда не показывают время, которое принадлежит далёкому прошлому или далёкому будущему (нет, по аналогии с предыдущими пунктами);
- у времени нет начала и конца (нет, у Unixtime, например, есть границы);
- одна минута на сервере, на котором работает база данных, имеет в точности такую же продолжительность, как одна минута на любых других часах (нет, часы на сервере могут «спешить» или «отставать», что приведёт к разной продолжительности временных отрезков на сервере и на эталонных часах);
- минимальной единицей времени является одна миллисекунда (нет, бывает необходимость хранить время с более высокой точностью);
- у формата времени всегда будет одинаковая степень точности (нет, вполне может получиться так, что придётся сравнивать дату (например, 2010-01-02) с датой и временем (например, 2010-01-02 12:34:55) и с временем с высокой точностью (например, 12:34:55.365245).
- метка времени достаточной точности можно считать уникальной (нет, ничто в реальности не мешает двум событиям произойти одновременно с точностью до, например, триллиардных долей секунды);
- метка времени показывает время, когда событие действительно произошло (нет, ничто не мешает сегодня опубликовать новость о событии вчерашнего дня);
- человекочитаемый формат даты и времени всегда одинаков (нет, например, «американский» формат («месяц/день/год») и «европейский» формат («день.месяц.год»);
- разность времени между двумя часовыми поясами будет оставаться постоянной (нет, периодически в разных странах принимают решения об изменении такой разницы);
- переход на летнее время происходит каждый год в одно и то же время (нет, а иногда и вообще не происходит, если в какой-то стране решили отменить этот переход);
- легко сосчитать количество часов и минут, прошедших с какого-то определённого момента времени (нет, если за прошедший период времени были переходы на летнее/зимнее время, смены часовых поясов и т.д.);
- время в Unixtime представляет собой количество секунд, начиная с 01 января 1970 года (нет, это верно только для часового пояса UTC+0, в остальных надо учитывать временную зону);
- можно подождать, когда часы покажут точно ЧЧ:ММ:СС с дискретизацией один раз в секунду (нет, вполне может случиться так, что в нужный момент времени операционная система не передаст вашей программе управление, и вы «пропустите» нужный момент);

- неделя начинается в понедельник (нет, во многих странах — в воскресенье);
- каждая минута содержит 60 секунд (нет, как минимум в силу периодических корректировок времени[260]);
- время всегда идёт вперёд (нет, например, в силу корректировки «спешащих» часов время на них периодически может «возвращаться назад»);
- 24:12:34 — неправильное время (нет, подобным образом иногда указывают время в аэропортах и на вокзалах);
- часовые пояса всегда различаются на целый час (нет, есть пояса с другим смещением);
- високосные годы наступают каждые 4 года (нет, здесь всё немного сложнее[261]).

Продолжим именами. Пусть с ними ситуация и немного проще, но — все эти утверждения также ложны:

- у каждого человека есть одно каноническое полное имя (нет, нужно учитывать национальные особенности);
- у каждого человека есть в точности N имён, независимо от значения N (нет, например, псевдонимы творческих людей);
- имена не меняются (нет, меняются — причём не только фамилии, но и имена — это разрешено законодательно в большинстве стран);
- имена записаны в какой-нибудь одной кодировке (нет, нужно учитывать национальные особенности);
- иногда в именах встречаются префиксы или суффиксы, но вы можете безопасно их игнорировать (нет, в юридическом контексте важен каждый символ);
- имена не содержат цифр (нет, в некоторых странах можно официально вписывать в имена цифры);
- имя и фамилия обязательно отличаются (нет, особенно в англоговорящих странах);
- две различные системы, в которых указано имя одного и того же человека, будут использовать для него одно и то же имя (нет, т.к. нет никаких законов физики, которые заставили бы эти системы синхронизироваться);
- если мы видим два разных ФИО — это разные люди (нет, например, человек поменял фамилию (и имя));
- если человек не менял имя и фамилию, его ФИО везде будет записано одинаково (нет, известны случаи, когда в свидетельстве о рождении, в паспорте и в водительском удостоверении у одного и того же человека ФИО было написано с опечатками — везде разными);
- у человека всегда есть имя (нет, например, у новорождённого).

Переходим к телефонным номерам. Все представленные ниже утверждения — также ложны:

- телефонный номер определённого типа (например, мобильный) никогда не сменит тип (нет, диапазон номеров могут передать другому оператору, и он вполне может использовать этот диапазон для стационарных телефонов);
- телефонный номер однозначно идентифицирует человека (нет, это может быть рабочий номер, по которому в разное время отвечают разные люди);
- у человека есть только один телефонный номер (нет, у многих есть несколько sim-карт);
- телефонные номера не могут быть использованы заново (нет, операторы связи выдают новым абонентам имена даже умерших людей);
- каждый код страны соответствует в точности одной стране (нет, например, у США, и Канады одинаковый код «+1», а у России и Казахстана — «+7»);
- каждой стране соответствует только один код (нет, есть страны с несколькими кодами);
- есть только два способа указания телефонного номера — в международном формате и в местном (нет, для некоторых номеров требуются различные префиксы, в зависимости от того, откуда набирается номер);

---

260 «Дополнительная секунда» («Wikipedia») [https://ru.wikipedia.org/wiki/Дополнительная_секунда]

261 «Високосный год» («Wikipedia») [https://ru.wikipedia.org/wiki/Високосный_год]

- ни один префикс реального телефонного номера не может быть реальным телефонным номером (нет, в некоторых странах можно попасть на другого абонента, если набрать дополнительные цифры после телефонного номера: например, номер «12345678» может принадлежать одному человеку, а номер «123456» — другому);
- телефонный номер содержит только цифры (нет, в Израиле, например, некоторые рекламные номера начинаются с «*»).

Теперь поговорим об адресах. Как легко догадаться, все представленные ниже утверждения также ложны:

- у каждого строения есть адрес (нет, система адресации не идеальная, и на планете полно строений без адреса);
- у каждого строения есть не более одного адреса (нет, у домов на перекрёстках улиц часто бывает два адреса);
- у каждого строения есть не более двух адресов (нет, бывает и перекрёсток трёх улиц, к тому же может понадобиться учитывать историческую хронику, когда менялись названия улиц);
- каждое строение находится в населённом пункте (нет, есть много строений в отдалённых районах, не относящихся к населённым пунктам);
- номер дома — это число (нет, у многих домов номера имеют буквенные префиксы или суффиксы вида «11а»);
- ноль в начале номера дома можно игнорировать (нет, есть города, где дома, например, «11» и «011» — это разные дома);
- в городе не может быть двух одноимённых улиц (нет, может, например, в Москве есть две улицы «8 марта»);
- бывают только улицы и проспекты (нет, бывают ещё площади, набережные и т.д.);
- не может так быть, чтобы у дома был номер корпуса, но не было номера дома (нет, может, пусть это и непривычно).

Этот список ложных утверждений — далеко не полный, но он даёт достаточное представление о том, что проектирование базы данных по принципу «я точно знаю, как бывает и как не бывает», скорее всего, принесёт в будущем вашим пользователям много проблем.

Именно поэтому необходимо тщательное изучение предметной области и всех требований заказчика и потребностей пользователей, чтобы спроектировать по-настоящему функциональную, надёжную, удобную для использования базу данных.

Задание 6.3.a: дополните представленные в данной главе списки теми типичными заблуждениями, которые вам известны.

Задание 6.2.b: существуют ли в базе данных «Банк»{408} и базе данных файлообменного сервиса{284}, {302}, {314} недоработки, вызванные следованием представленным в данной главе заблуждениям? Составьте список таких недоработок.

6.3.2. ТИПИЧНЫЕ ОШИБКИ ПРИ РАБОТЕ С БАЗАМИ ДАННЫХ И СПОСОБЫ ИХ УСТРАНЕНИЯ

В данной (финальной) главе книги собраны воедино наиболее типичные и опасные из ранее упомянутых проблем и ошибок, а также приведено несколько дополнительных рекомендаций.

Начнём с проблем, уже рассмотренных ранее в данной книге.

Проблема	Решение
Автоконвертация моделей базы данных{28}. Автоматическая конвертация даталогической модели базы данных в физическую, как правило, приводит к нарушению удобства использования{14}, производительности{16} и защищённости данных{17}.	Только вдумчивое «ручное» создание физической модели позволяет получить по-настоящему качественную базу данных, отвечающую всем необходимым требованиям. Мнимая экономия времени на автоматическим создании модели потом выльется в тысячи очень серьёзных проблем.
Игнорирование проблем с удобством использования базы данных{16}. Как правило, проблемы с удобством использования{14} приводят к увеличению количества проблем с производительностью, т.к. сложные запросы тяжелее оптимизировать.	Как только для выполнения простых, типичных, тривиальных операций приходится писать сложные запросы (или просто количество сложных запросов начинает ощутимо возрастать), стоит пересмотреть модель базы данных с тем, чтобы оптимизировать её с точки зрения удобства использования{14}.
Игнорирование реализации ограничений на уровне базы данных{217}. Подавляющее большинство катастрофических проблем с базами данных происходит именно потому, что кто-то когда-то посчитал ту или иную фатальную ситуацию маловероятной или вовсе невозможной и не реализовал защитный механизм.	Если вы понимаете, что некое ограничение необходимо, его стоит реализовать средствами базы данных или СУБД, не надеясь, что кто-то где-то сделает это за вас.
Использование «не английского» языка{27}. Несмотря на то, что подавляющее большинство средств проектирования баз данных и СУБД поддерживают возможность именования объектов на русском языке (и многих других), всегда может возникнуть ситуация, в которой вся система перестанет работать из-за проблем с кодировками в одном из множества программных средств.	Всегда используйте для именования объектов базы данных только английский язык. Если вы его не знаете, пишите «транслитом», но всё равно используйте только английский алфавит.
Использование первого же сработавшего решения{392}. Прекращение анализа проблемы после того, как найдено первое же работающее решение.	Поскольку такое решение может не быть самым лучшим или может покрывать лишь частные случаи поставленной задачи, стоит всегда выполнять поиск нескольких альтернативных решений и выбор наилучшего из них.

Нарушение единого стиля именования объектов базы данных{27}. В масштабах модели всей базы данных проблема несоблюдения стандартов именования превращается в катастрофу: в такой «мешанине» очень тяжело ориентироваться, очень легко пропустить ошибку.	Следует выработать стандарт именования объектов базы данных и строго ему следовать.
Нарушение единства смысла данных в столбце{99}. При хранении или выборке данных содержимое одного и того же столбца имеет разный смысл в разных строках (например, в некоем столбце `measurement` одной строке хранится рост человека, а в другой — вес).	По определению данные в одном столбце таблицы (атрибуте отношения) должны иметь один и тот же смысл, т.е. одинаково трактоваться для всех записей таблицы (кортежей отношения). Следует избегать написания запросов или создания таких структур базы данных, в которых это правило нарушалось бы.
Нарушение последовательности полей в составных ключах и индексах{108}. Неверная последовательность полей в составных ключах и индексах ощутимо снижает производительность.	То поле в составном ключе или индексе, по которому поиск часто будет выполняться отдельно от других полей, должно быть первым.
Неверная трактовка зависимостей{174}. Попытка представить (обнаружить) зависимости{174} в отношениях вне контекста предметной области, т.е. как некую универсальную абстракцию, справедливую для всех случаев её применения.	Наличие или отсутствие в отношении той или иной зависимости всецело завязано на требования предметной области, которые обязательно необходимо учитывать при анализе зависимостей и нормализации схем отношений.
Недостаточное внимание кодировкам{316}. Если вы полагаетесь на настройки кодировок по умолчанию, с очень высокой вероятностью ваша база данных будет работать на каком-то из серверов неверно.	Необходимо явным образом указывать все настройки кодировок везде, где это технически возможно.
Недостаточный анализ предметной области{387}. Неполный анализ предметной области приводит к отсутствию в базе данных той или иной необходимой информации.	Необходимо тщательно выяснить (у заказчика, из стандартов, из общения с пользователями), какие данные необходимы для полноценной работы с базой данных, и создать соответствующие структуры для хранения этих данных.

Если же говорить об общих рекомендациях, которых также уже было приведено немало, можно сформировать такой список — не столько идей, сколько финальных напутствий.

Помните о цели использования данных. Данные всегда хранятся для последующего эффективного использования. Потому важно понимать, какие операции над данными и с какой частотой будут выполняться. Такое понимание позволяет ещё на стадии проектирования повысить удобство{14} и производительность{16} базы данных.

Подходите к нормализации{211} **и иным техникам разумно**. Как недостаточная, так и избыточная нормализация приводит к серии очень неприятных последствий. Золотую середину найти сложно, но если у вас есть сомнения — сначала нормализуйте (выполнить обратный процесс быстрее и проще).

Это же относится и к любым другим техникам — они не должны становиться самоцелью. Они — лишь инструменты, помогающие в решении поставленных задач.

Принимайте во внимание возможности СУБД. Иногда некоторых изощрённых решений удаётся избежать потому, что на уровне СУБД уже есть готовые механизмы, позволяющие достичь поставленных целей. И даже простое понимание типичных возможностей СУБД очень облегчает разработку, ведь:

- представления{340} позволяют избежать денормализации на уровне таблиц;
- индексы{106} повышают производительность;
- проверки{347} упрощают формирование специфических ограничений;
- хранимые подпрограммы{363} позволяет на уровне базы данных один раз реализовать то, что, возможно, пришлось бы реализовывать в нескольких приложениях;
- управление транзакциями{374} позволяет получить в точности то поведение базы данных, которое вам требуется;
- триггеры{350} позволяют автоматически выполнять целый спектр полезных (и, порой, необходимых) операций.

Подходите к проектированию серьёзно. Предполагается, что вы создаёте базу данных, с которой многие люди будут работать на протяжении долгого времени — и не только в качестве конечных пользователей.

Следование соглашению об именовании{218}, полноценное документирование{220}, использование общепринятых средств проектирования{220} и иные очевидные техники достаточно просты в использовании, и в то же время ощутимо повышают качество вашей разработки.

Помните об архивировании и протоколировании. Сложные базы данных требуют особого подхода к внутренней структуре — как правило, большая часть данных будет востребована крайне редко и может быть перенесена в специальные архивные таблицы по мере своего «устаревания».

Вместо удаления информации можно для повышения надёжности использовать т.н. «мягкое удаление», когда данные лишь помечаются как удалённые, но физически остаются в базе данных.

А полноценный механизм протоколирования всех критических действий поможет не только выяснить причину возникновения той или иной ситуации, но и восстановить исходное состояние базы данных при необходимости.

Используйте знания из смежных областей. Несмотря на то, что проектирование баз данных само по себе является обширной и сложной областью, требующей глубокого понимания (и соответствующей специализации), знания языка SQL, понимание принципов работы операционных систем и компьютерных сетей, опыт работы с несколькими различными СУБД, хотя бы некоторые представления о безопасности — всё это и многое другое позволит вам более целостно подходить к проектированию баз данных, учитывать намного больше различных ситуаций, предвидеть множество проблем и эффективно защищаться от различных неприятностей ещё до того, как они возникнут.

На этом — всё. Успешного проектирования баз данных!

ПРИЛОЖЕНИЯ

ОПИСАНИЕ БАЗЫ ДАННЫХ ДЛЯ ВЫПОЛНЕНИЯ САМОСТОЯТЕЛЬНЫХ ЗАДАНИЙ

Для выполнения многих самостоятельных заданий, представленных в данной книге, вам понадобится база данных, описание которой приведено в этом приложении.

Это предельно упрощённая учебная база данных гипотетического банка (далее — база данных «Банк»), которая содержит в себе только самый необходимый минимум элементов.

В этой базе данных осознанно допущено некоторое количество ошибок, обнаружить и устранить которые вам предлагается в различных заданиях, представленных в данной книге.

База данных отражает следующие сущности и их атрибуты (рисунок А.1.а):

- Счёт (описывает банковский счёт):
  - идентификатор (идентификатор счёта);
  - баланс (баланс счёта, в денежных единицах);
  - владелец (ссылка на владельца счёта);
  - системный счёт (признак того, что данный счёт не принадлежит человеку).
- Статус (описывает статус счёта, например, «активный», «заблокирован» и т.д.):
  - идентификатор (идентификатор статуса);
  - имя (наименование статуса).
- Платёж операционный (описывает платежи, проведённые в текущем месяце):
  - идентификатор (идентификатор платежа);
  - счёт-источник (ссылка на счёт-источник);
  - счёт-приёмник (ссылка на счёт-приёмник);
  - дата и время (дата и время проведения платежа);
  - сумма (сумма платежа).
- Платёж архивный (описывает платежи, проведённые до начала текущего месяца):
  - идентификатор (идентификатор платежа);
  - счёт-источник (ссылка на счёт-источник);
  - счёт-приёмник (ссылка на счёт-приёмник);

    - дата и время (дата и время проведения платежа);
    - сумма (сумма платежа).
- Владелец счёта (описывает владельца счёта):
    - идентификатор (идентификатор владельца счёта);
    - имя (имя владельца счёта).
- Страница сайта (описывает страницу сайта банка):
    - идентификатор (идентификатор страницы);
    - родительская страница (ссылка на родительскую страницу);
    - имя (имя страницы).
- Офис (описывает офис банка):
    - идентификатор (идентификатор офиса);
    - город (город, в котором находится офис);
    - название (название офиса);
    - сумма продаж (суммарная прибыль от услуг, оказанных офисом).

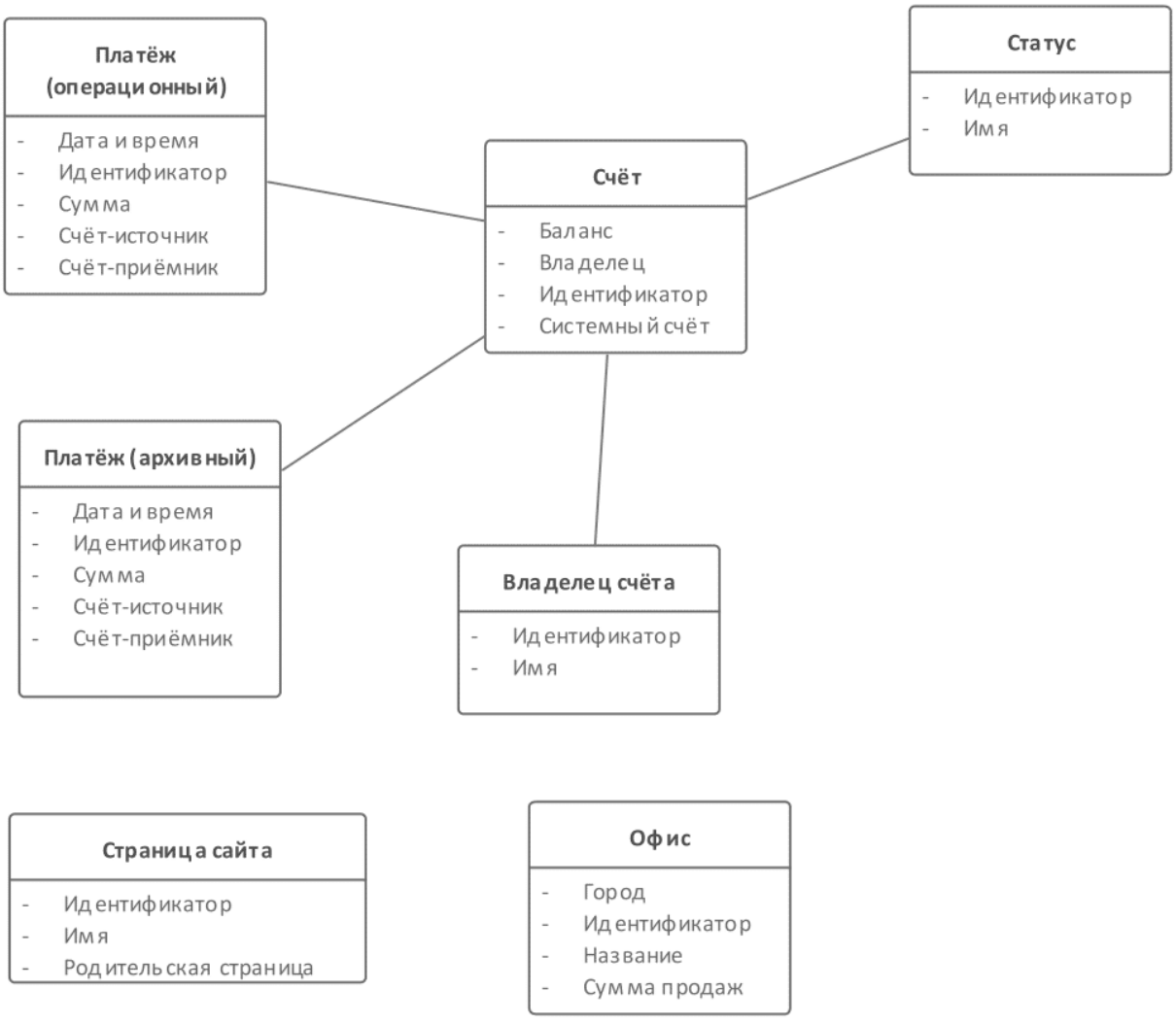


*Рисунок А.1.а — **Инфологическая модель базы данных «Банк»***

Даталогические модели в привязке к MS SQL Server и Oracle представлены на рисунках A.1.b и A.1.c соответственно. Если вам удобнее работать с моделью для MySQL, вы можете подготовить её самостоятельно на основе раздаточного материала{5}. Там же вы можете найти исходный проект базы данных «Банк» в формате Sparx Enterprise Architect.

В качестве описания физического уровня базы данных вы можете рассмотреть код, представленный в следующем приложении.

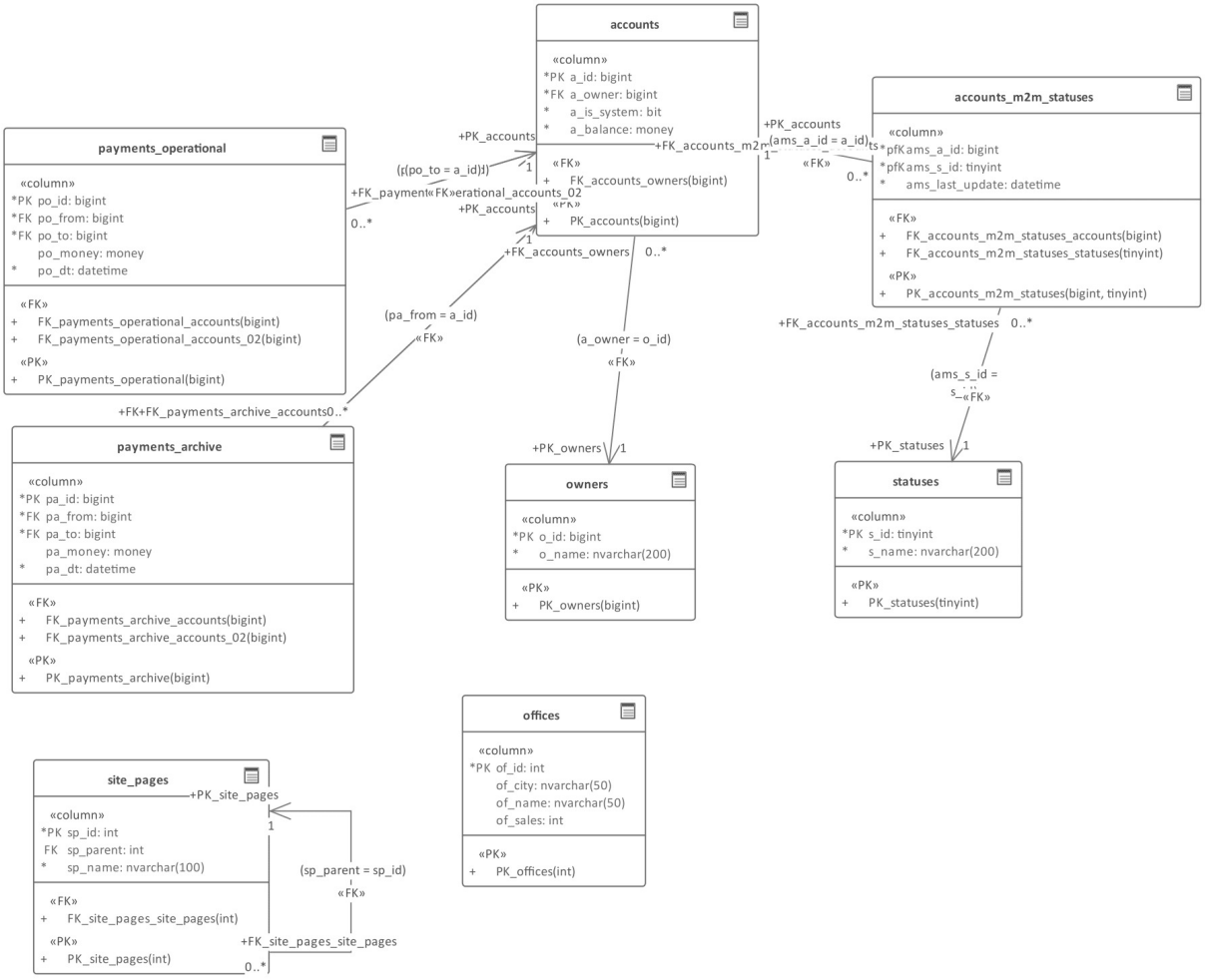


*Рисунок A.1.b — **Даталогическая модель базы данных «Банк» для MS SQL Server***

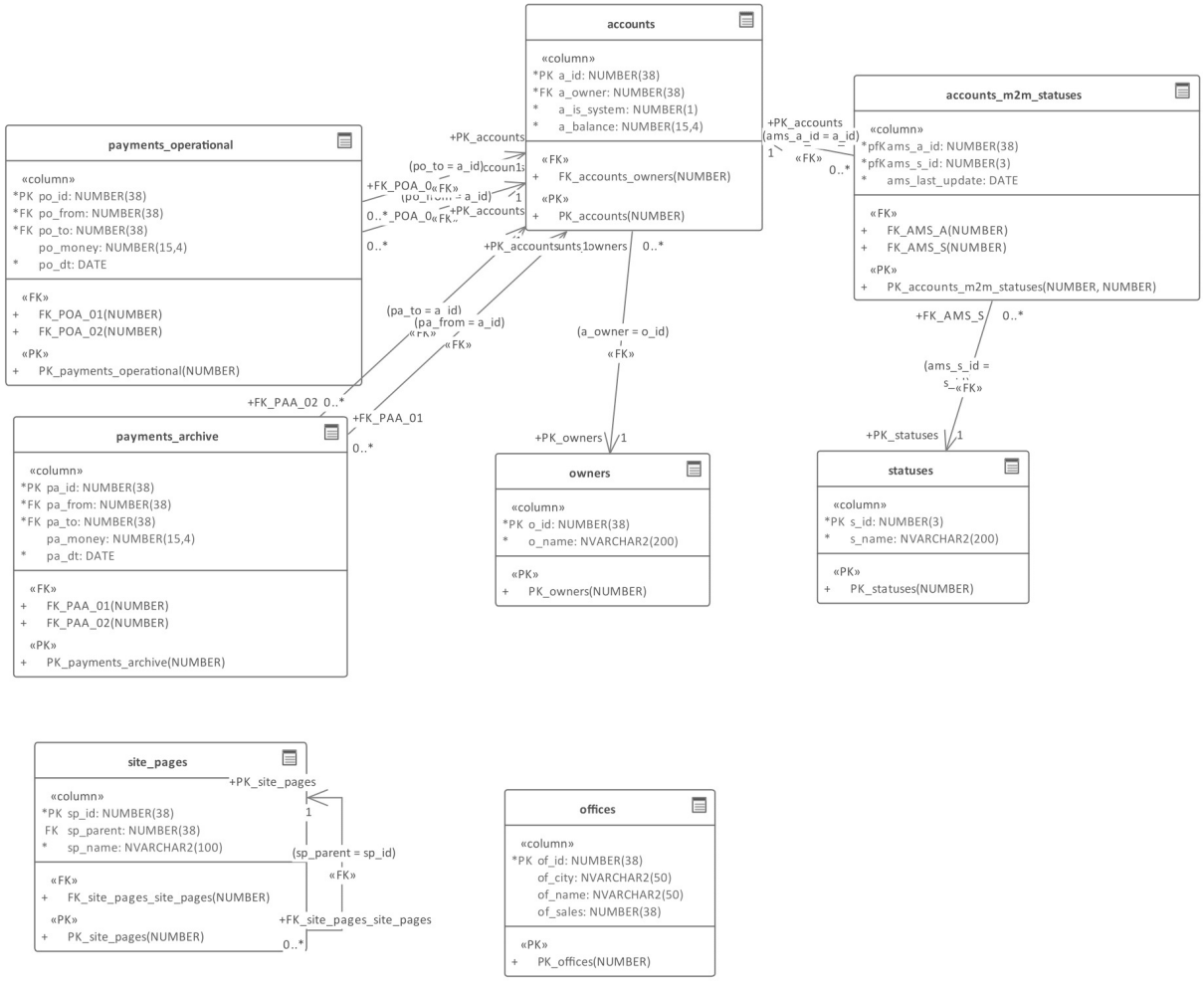


*Рисунок А.1.с — **Даталогическая модель базы данных «Банк» для Oracle***

7.2. КОД БАЗЫ ДАННЫХ ДЛЯ ВЫПОЛНЕНИЯ САМОСТОЯТЕЛЬНЫХ ЗАДАНИЙ

В данном приложении «на самый крайний случай» приведён SQL-код для создания базы данных «Банк».

Код для MS SQL Server:

```
IF EXISTS (SELECT * FROM dbo.sysobjects WHERE id = object_id('[FK_site_pages_site_pages]') AND
OBJECTPROPERTY(id, 'IsForeignKey') = 1)
ALTER TABLE [site_pages] DROP CONSTRAINT [FK_site_pages_site_pages]
GO

IF EXISTS (SELECT * FROM dbo.sysobjects WHERE id = object_id('[FK_payments_operational_accounts]') AND
OBJECTPROPERTY(id, 'IsForeignKey') = 1)
ALTER TABLE [payments_operational] DROP CONSTRAINT [FK_payments_operational_accounts]
GO

IF EXISTS (SELECT * FROM dbo.sysobjects WHERE id = object_id('[FK_payments_operational_accounts_02]') AND
OBJECTPROPERTY(id, 'IsForeignKey') = 1)
ALTER TABLE [payments_operational] DROP CONSTRAINT [FK_payments_operational_accounts_02]
GO

IF EXISTS (SELECT * FROM dbo.sysobjects WHERE id = object_id('[FK_payments_archive_accounts]') AND
OBJECTPROPERTY(id, 'IsForeignKey') = 1)
ALTER TABLE [payments_archive] DROP CONSTRAINT [FK_payments_archive_accounts]
GO

IF EXISTS (SELECT * FROM dbo.sysobjects WHERE id = object_id('[FK_payments_archive_accounts_02]') AND
OBJECTPROPERTY(id, 'IsForeignKey') = 1)
ALTER TABLE [payments_archive] DROP CONSTRAINT [FK_payments_archive_accounts_02]
GO

IF EXISTS (SELECT * FROM dbo.sysobjects WHERE id = object_id('[FK_accounts_m2m_statuses_accounts]') AND
OBJECTPROPERTY(id, 'IsForeignKey') = 1)
ALTER TABLE [accounts_m2m_statuses] DROP CONSTRAINT [FK_accounts_m2m_statuses_accounts]
GO

IF EXISTS (SELECT * FROM dbo.sysobjects WHERE id = object_id('[FK_accounts_m2m_statuses_statuses]') AND
OBJECTPROPERTY(id, 'IsForeignKey') = 1)
ALTER TABLE [accounts_m2m_statuses] DROP CONSTRAINT [FK_accounts_m2m_statuses_statuses]
GO

IF EXISTS (SELECT * FROM dbo.sysobjects WHERE id = object_id('[FK_accounts_owners]') AND
OBJECTPROPERTY(id, 'IsForeignKey') = 1)
ALTER TABLE [accounts] DROP CONSTRAINT [FK_accounts_owners]
GO

IF EXISTS (SELECT * FROM dbo.sysobjects WHERE id = object_id('[statuses]') AND OBJECTPROPERTY(id,
'IsUserTable') = 1)
DROP TABLE [statuses]
GO

IF EXISTS (SELECT * FROM dbo.sysobjects WHERE id = object_id('[site_pages]') AND OBJECTPROPERTY(id,
'IsUserTable') = 1)
DROP TABLE [site_pages]
GO

IF EXISTS (SELECT * FROM dbo.sysobjects WHERE id = object_id('[payments_operational]') AND
OBJECTPROPERTY(id, 'IsUserTable') = 1)
DROP TABLE [payments_operational]
GO

IF EXISTS (SELECT * FROM dbo.sysobjects WHERE id = object_id('[payments_archive]') AND OBJECTPROPERTY(id,
'IsUserTable') = 1)
```

```
DROP TABLE [payments_archive]
GO

IF EXISTS (SELECT * FROM dbo.sysobjects WHERE id = object_id('[owners]') AND OBJECTPROPERTY(id,
'IsUserTable') = 1)
DROP TABLE [owners]
GO

IF EXISTS (SELECT * FROM dbo.sysobjects WHERE id = object_id('[offices]') AND OBJECTPROPERTY(id,
'IsUserTable') = 1)
DROP TABLE [offices]
GO

IF EXISTS (SELECT * FROM dbo.sysobjects WHERE id = object_id('[accounts_m2m_statuses]') AND
OBJECTPROPERTY(id, 'IsUserTable') = 1)
DROP TABLE [accounts_m2m_statuses]
GO

IF EXISTS (SELECT * FROM dbo.sysobjects WHERE id = object_id('[accounts]') AND OBJECTPROPERTY(id,
'IsUserTable') = 1)
DROP TABLE [accounts]
GO

CREATE TABLE [statuses]
(
        [s_id] tinyint NOT NULL IDENTITY (1, 1),
        [s_name] nvarchar(200) NOT NULL
)
GO

CREATE TABLE [site_pages]
(
        [sp_id] int NOT NULL IDENTITY (1, 1),
        [sp_parent] int NULL,
        [sp_name] nvarchar(100) NOT NULL
)
GO

CREATE TABLE [payments_operational]
(
        [po_id] bigint NOT NULL IDENTITY (1, 1),
        [po_from] bigint NOT NULL,
        [po_to] bigint NOT NULL,
        [po_money] money NULL,
        [po_dt] datetime NOT NULL
)
GO

CREATE TABLE [payments_archive]
(
        [pa_id] bigint NOT NULL,
        [pa_from] bigint NOT NULL,
        [pa_to] bigint NOT NULL,
        [pa_money] money NULL,
        [pa_dt] datetime NOT NULL
)
GO

CREATE TABLE [owners]
(
        [o_id] bigint NOT NULL IDENTITY (1, 1),
        [o_name] nvarchar(200) NOT NULL
)
GO

CREATE TABLE [offices]
(
        [of_id] int NOT NULL IDENTITY (1, 1),
        [of_city] nvarchar(50) NULL,
        [of_name] nvarchar(50) NULL,
        [of_sales] int NULL
)
GO

CREATE TABLE [accounts_m2m_statuses]
(
        [ams_a_id] bigint NOT NULL,
        [ams_s_id] tinyint NOT NULL,
        [ams_last_update] datetime NOT NULL
)
GO
```

```
CREATE TABLE [accounts]
(
        [a_id] bigint NOT NULL IDENTITY (1, 1),
        [a_owner] bigint NOT NULL,
        [a_is_system] bit NOT NULL,
        [a_balance] money NOT NULL
)
GO

ALTER TABLE [statuses]
 ADD CONSTRAINT [PK_statuses]
        PRIMARY KEY CLUSTERED ([s_id])
GO

ALTER TABLE [site_pages]
 ADD CONSTRAINT [PK_site_pages]
        PRIMARY KEY CLUSTERED ([sp_id])
GO

ALTER TABLE [payments_operational]
 ADD CONSTRAINT [PK_payments_operational]
        PRIMARY KEY CLUSTERED ([po_id])
GO

ALTER TABLE [payments_archive]
 ADD CONSTRAINT [PK_payments_archive]
        PRIMARY KEY CLUSTERED ([pa_id])
GO

ALTER TABLE [owners]
 ADD CONSTRAINT [PK_owners]
        PRIMARY KEY CLUSTERED ([o_id])
GO

ALTER TABLE [offices]
 ADD CONSTRAINT [PK_offices]
        PRIMARY KEY CLUSTERED ([of_id])
GO

ALTER TABLE [accounts_m2m_statuses]
 ADD CONSTRAINT [PK_accounts_m2m_statuses]
        PRIMARY KEY CLUSTERED ([ams_a_id],[ams_s_id])
GO

ALTER TABLE [accounts]
 ADD CONSTRAINT [PK_accounts]
        PRIMARY KEY CLUSTERED ([a_id])
GO

ALTER TABLE [site_pages] ADD CONSTRAINT [FK_site_pages_site_pages]
        FOREIGN KEY ([sp_parent]) REFERENCES [site_pages] ([sp_id]) ON DELETE No Action ON UPDATE No
Action
GO

ALTER TABLE [payments_operational] ADD CONSTRAINT [FK_payments_operational_accounts]
        FOREIGN KEY ([po_from]) REFERENCES [accounts] ([a_id]) ON DELETE No Action ON UPDATE No Action
GO

ALTER TABLE [payments_operational] ADD CONSTRAINT [FK_payments_operational_accounts_02]
        FOREIGN KEY ([po_to]) REFERENCES [accounts] ([a_id]) ON DELETE No Action ON UPDATE No Action
GO

ALTER TABLE [payments_archive] ADD CONSTRAINT [FK_payments_archive_accounts]
        FOREIGN KEY ([pa_from]) REFERENCES [accounts] ([a_id]) ON DELETE No Action ON UPDATE No Action
GO

ALTER TABLE [payments_archive] ADD CONSTRAINT [FK_payments_archive_accounts_02]
        FOREIGN KEY ([pa_to]) REFERENCES [accounts] ([a_id]) ON DELETE No Action ON UPDATE No Action
GO

ALTER TABLE [accounts_m2m_statuses] ADD CONSTRAINT [FK_accounts_m2m_statuses_accounts]
        FOREIGN KEY ([ams_a_id]) REFERENCES [accounts] ([a_id]) ON DELETE No Action ON UPDATE No Action
GO

ALTER TABLE [accounts_m2m_statuses] ADD CONSTRAINT [FK_accounts_m2m_statuses_statuses]
        FOREIGN KEY ([ams_s_id]) REFERENCES [statuses] ([s_id]) ON DELETE No Action ON UPDATE No Action
GO

ALTER TABLE [accounts] ADD CONSTRAINT [FK_accounts_owners]
        FOREIGN KEY ([a_owner]) REFERENCES [owners] ([o_id]) ON DELETE No Action ON UPDATE No Action
GO
```

Код для Oracle:

```
DECLARE
  C NUMBER;
BEGIN
SELECT COUNT(*) INTO C
FROM ALL_TRIGGERS
  WHERE OWNER = ''
  AND TRIGGER_NAME = 'TRG_statuses_s_id';
  IF (C > 0) THEN
    EXECUTE IMMEDIATE 'DROP TRIGGER "TRG_statuses_s_id"';
END IF;
END
;

DECLARE
  C NUMBER;
BEGIN
SELECT COUNT(*) INTO C
FROM ALL_SEQUENCES
  WHERE SEQUENCE_OWNER = ''
  AND SEQUENCE_NAME = 'SEQ_statuses_s_id';
  IF (C > 0) THEN
    EXECUTE IMMEDIATE 'DROP SEQUENCE "SEQ_statuses_s_id"';
END IF;
END
;

DECLARE
  C NUMBER;
BEGIN
SELECT COUNT(*) INTO C
FROM ALL_TRIGGERS
  WHERE OWNER = ''
  AND TRIGGER_NAME = 'TRG_site_pages_sp_id';
  IF (C > 0) THEN
    EXECUTE IMMEDIATE 'DROP TRIGGER "TRG_site_pages_sp_id"';
END IF;
END
;

DECLARE
  C NUMBER;
BEGIN
SELECT COUNT(*) INTO C
FROM ALL_SEQUENCES
  WHERE SEQUENCE_OWNER = ''
  AND SEQUENCE_NAME = 'SEQ_site_pages_sp_id';
  IF (C > 0) THEN
    EXECUTE IMMEDIATE 'DROP SEQUENCE "SEQ_site_pages_sp_id"';
END IF;
END
;

DECLARE
  C NUMBER;
BEGIN
SELECT COUNT(*) INTO C
FROM ALL_TRIGGERS
  WHERE OWNER = ''
  AND TRIGGER_NAME = 'TRG_payments_operational_po_id';
  IF (C > 0) THEN
    EXECUTE IMMEDIATE 'DROP TRIGGER "TRG_payments_operational_po_id"';
END IF;
END
;

DECLARE
  C NUMBER;
```

```
BEGIN
SELECT COUNT(*) INTO C
FROM ALL_SEQUENCES
  WHERE SEQUENCE_OWNER = ''
  AND SEQUENCE_NAME = 'SEQ_payments_operational_po_id';
  IF (C > 0) THEN
    EXECUTE IMMEDIATE 'DROP SEQUENCE "SEQ_payments_operational_po_id"';
END IF;
END
;

DECLARE
  C NUMBER;
BEGIN
SELECT COUNT(*) INTO C
FROM ALL_TRIGGERS
  WHERE OWNER = ''
  AND TRIGGER_NAME = 'TRG_owners_o_id';
  IF (C > 0) THEN
    EXECUTE IMMEDIATE 'DROP TRIGGER "TRG_owners_o_id"';
END IF;
END
;

DECLARE
  C NUMBER;
BEGIN
SELECT COUNT(*) INTO C
FROM ALL_SEQUENCES
  WHERE SEQUENCE_OWNER = ''
  AND SEQUENCE_NAME = 'SEQ_owners_o_id';
  IF (C > 0) THEN
    EXECUTE IMMEDIATE 'DROP SEQUENCE "SEQ_owners_o_id"';
END IF;
END
;

DECLARE
  C NUMBER;
BEGIN
SELECT COUNT(*) INTO C
FROM ALL_TRIGGERS
  WHERE OWNER = ''
  AND TRIGGER_NAME = 'TRG_offices_of_id';
  IF (C > 0) THEN
    EXECUTE IMMEDIATE 'DROP TRIGGER "TRG_offices_of_id"';
END IF;
END
;

DECLARE
  C NUMBER;
BEGIN
SELECT COUNT(*) INTO C
FROM ALL_SEQUENCES
  WHERE SEQUENCE_OWNER = ''
  AND SEQUENCE_NAME = 'SEQ_offices_of_id';
  IF (C > 0) THEN
    EXECUTE IMMEDIATE 'DROP SEQUENCE "SEQ_offices_of_id"';
END IF;
END
;

DECLARE
  C NUMBER;
BEGIN
SELECT COUNT(*) INTO C
FROM ALL_TRIGGERS
  WHERE OWNER = ''
```

```
  AND TRIGGER_NAME = 'TRG_accounts_a_id';
  IF (C > 0) THEN
    EXECUTE IMMEDIATE 'DROP TRIGGER "TRG_accounts_a_id"';
END IF;
END
;

DECLARE
  C NUMBER;
BEGIN
SELECT COUNT(*) INTO C
FROM ALL_SEQUENCES
  WHERE SEQUENCE_OWNER = ''
  AND SEQUENCE_NAME = 'SEQ_accounts_a_id';
  IF (C > 0) THEN
    EXECUTE IMMEDIATE 'DROP SEQUENCE "SEQ_accounts_a_id"';
END IF;
END
;

DROP TABLE "statuses" CASCADE CONSTRAINTS
;

DROP TABLE "site_pages" CASCADE CONSTRAINTS
;

DROP TABLE "payments_operational" CASCADE CONSTRAINTS
;

DROP TABLE "payments_archive" CASCADE CONSTRAINTS
;

DROP TABLE "owners" CASCADE CONSTRAINTS
;

DROP TABLE "offices" CASCADE CONSTRAINTS
;

DROP TABLE "accounts_m2m_statuses" CASCADE CONSTRAINTS
;

DROP TABLE "accounts" CASCADE CONSTRAINTS
;

CREATE TABLE "statuses"
(
        "s_id" NUMBER(3) NOT NULL,
        "s_name" NVARCHAR2(200) NOT NULL
)
;

CREATE TABLE "site_pages"
(
        "sp_id" NUMBER(38) NOT NULL,
        "sp_parent" NUMBER(38) NULL,
        "sp_name" NVARCHAR2(100) NOT NULL
)
;

CREATE TABLE "payments_operational"
(
        "po_id" NUMBER(38) NOT NULL,
        "po_from" NUMBER(38) NOT NULL,
        "po_to" NUMBER(38) NOT NULL,
        "po_money" NUMBER(15,4) NULL,
        "po_dt" DATE NOT NULL
)
;
```

```
CREATE TABLE "payments_archive"
(
        "pa_id" NUMBER(38) NOT NULL,
        "pa_from" NUMBER(38) NOT NULL,
        "pa_to" NUMBER(38) NOT NULL,
        "pa_money" NUMBER(15,4) NULL,
        "pa_dt" DATE NOT NULL
)
;

CREATE TABLE "owners"
(
        "o_id" NUMBER(38) NOT NULL,
        "o_name" NVARCHAR2(200) NOT NULL
)
;

CREATE TABLE "offices"
(
        "of_id" NUMBER(38) NOT NULL,
        "of_city" NVARCHAR2(50) NULL,
        "of_name" NVARCHAR2(50) NULL,
        "of_sales" NUMBER(38) NULL
)
;

CREATE TABLE "accounts_m2m_statuses"
(
        "ams_a_id" NUMBER(38) NOT NULL,
        "ams_s_id" NUMBER(3) NOT NULL,
        "ams_last_update" DATE NOT NULL
)
;

CREATE TABLE "accounts"
(
        "a_id" NUMBER(38) NOT NULL,
        "a_owner" NUMBER(38) NOT NULL,
        "a_is_system" NUMBER(1) NOT NULL,
        "a_balance" NUMBER(15,4) NOT NULL
)
;

CREATE SEQUENCE "SEQ_statuses_s_id"
        INCREMENT BY 1
        START WITH 1
        NOMAXVALUE
        MINVALUE  1
        NOCYCLE
        NOCACHE
        NOORDER
;

CREATE OR REPLACE TRIGGER "TRG_statuses_s_id"
        BEFORE INSERT
        ON "statuses"
        FOR EACH ROW
        BEGIN
                SELECT "SEQ_statuses_s_id".NEXTVAL
                INTO :NEW."s_id"
                FROM DUAL;
        END;
;
/


CREATE SEQUENCE "SEQ_site_pages_sp_id"
        INCREMENT BY 1
        START WITH 1
```

```
        NOMAXVALUE
        MINVALUE  1
        NOCYCLE
        NOCACHE
        NOORDER
;

CREATE OR REPLACE TRIGGER "TRG_site_pages_sp_id"
        BEFORE INSERT
        ON "site_pages"
        FOR EACH ROW
        BEGIN
                SELECT "SEQ_site_pages_sp_id".NEXTVAL
                INTO :NEW."sp_id"
                FROM DUAL;
        END;
;
/


CREATE SEQUENCE "SEQ_payments_operational_po_id"
        INCREMENT BY 1
        START WITH 1
        NOMAXVALUE
        MINVALUE  1
        NOCYCLE
        NOCACHE
        NOORDER
;

CREATE OR REPLACE TRIGGER "TRG_payments_operational_po_id"
        BEFORE INSERT
        ON "payments_operational"
        FOR EACH ROW
        BEGIN
                SELECT "SEQ_payments_operational_po_id".NEXTVAL
                INTO :NEW."po_id"
                FROM DUAL;
        END;
;
/

CREATE SEQUENCE "SEQ_owners_o_id"
        INCREMENT BY 1
        START WITH 1
        NOMAXVALUE
        MINVALUE  1
        NOCYCLE
        NOCACHE
        NOORDER
;

CREATE OR REPLACE TRIGGER "TRG_owners_o_id"
        BEFORE INSERT
        ON "owners"
        FOR EACH ROW
        BEGIN
                SELECT "SEQ_owners_o_id".NEXTVAL
                INTO :NEW."o_id"
                FROM DUAL;
        END;
;
/


CREATE SEQUENCE "SEQ_offices_of_id"
        INCREMENT BY 1
        START WITH 1
        NOMAXVALUE
```

```
        MINVALUE  1
        NOCYCLE
        NOCACHE
        NOORDER
;

CREATE OR REPLACE TRIGGER "TRG_offices_of_id"
        BEFORE INSERT
        ON "offices"
        FOR EACH ROW
        BEGIN
                SELECT "SEQ_offices_of_id".NEXTVAL
                INTO :NEW."of_id"
                FROM DUAL;
        END;
;
/

CREATE SEQUENCE "SEQ_accounts_a_id"
        INCREMENT BY 1
        START WITH 1
        NOMAXVALUE
        MINVALUE  1
        NOCYCLE
        NOCACHE
        NOORDER
;

CREATE OR REPLACE TRIGGER "TRG_accounts_a_id"
        BEFORE INSERT
        ON "accounts"
        FOR EACH ROW
        BEGIN
                SELECT "SEQ_accounts_a_id".NEXTVAL
                INTO :NEW."a_id"
                FROM DUAL;
        END;
;
/

ALTER TABLE "statuses"
 ADD CONSTRAINT "PK_statuses"
        PRIMARY KEY ("s_id") USING INDEX
;

ALTER TABLE "site_pages"
 ADD CONSTRAINT "PK_site_pages"
        PRIMARY KEY ("sp_id") USING INDEX
;

ALTER TABLE "payments_operational"
 ADD CONSTRAINT "PK_payments_operational"
        PRIMARY KEY ("po_id") USING INDEX
;

ALTER TABLE "payments_archive"
 ADD CONSTRAINT "PK_payments_archive"
        PRIMARY KEY ("pa_id") USING INDEX
;

ALTER TABLE "owners"
 ADD CONSTRAINT "PK_owners"
        PRIMARY KEY ("o_id") USING INDEX
;

ALTER TABLE "offices"
 ADD CONSTRAINT "PK_offices"
        PRIMARY KEY ("of_id") USING INDEX
;
```

```
ALTER TABLE "accounts_m2m_statuses"
 ADD CONSTRAINT "PK_accounts_m2m_statuses"
        PRIMARY KEY ("ams_a_id","ams_s_id") USING INDEX
;

ALTER TABLE "accounts"
 ADD CONSTRAINT "PK_accounts"
        PRIMARY KEY ("a_id") USING INDEX
;

ALTER TABLE "site_pages"
 ADD CONSTRAINT "FK_site_pages_site_pages"
        FOREIGN KEY ("sp_parent") REFERENCES "site_pages" ("sp_id")
;

ALTER TABLE "payments_operational"
 ADD CONSTRAINT "FK_POA_01"
        FOREIGN KEY ("po_from") REFERENCES "accounts" ("a_id")
;

ALTER TABLE "payments_operational"
 ADD CONSTRAINT "FK_POA_02"
        FOREIGN KEY ("po_to") REFERENCES "accounts" ("a_id")
;

ALTER TABLE "payments_archive"
 ADD CONSTRAINT "FK_PAA_01"
        FOREIGN KEY ("pa_from") REFERENCES "accounts" ("a_id")
;

ALTER TABLE "payments_archive"
 ADD CONSTRAINT "FK_PAA_02"
        FOREIGN KEY ("pa_to") REFERENCES "accounts" ("a_id")
;

ALTER TABLE "accounts_m2m_statuses"
 ADD CONSTRAINT "FK_AMS_A"
        FOREIGN KEY ("ams_a_id") REFERENCES "accounts" ("a_id")
;

ALTER TABLE "accounts_m2m_statuses"
 ADD CONSTRAINT "FK_AMS_S"
        FOREIGN KEY ("ams_s_id") REFERENCES "statuses" ("s_id")
;

ALTER TABLE "accounts"
 ADD CONSTRAINT "FK_accounts_owners"
        FOREIGN KEY ("a_owner") REFERENCES "owners" ("o_id")
```

ЛИЦЕНЗИЯ И РАСПРОСТРАНЕНИЕ

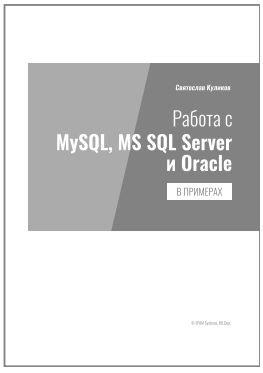
Текст книги периодически обновляется и дорабатывается. Если вы хотите поделиться этой книгой, пожалуйста, делитесь ссылкой на самую актуальную версию, доступную здесь: http://svyatoslav.biz/relational_databases_book_download/.

Исходные материалы (схемы, скрипты и т.д.) можно получить по этой ссылке: http://svyatoslav.biz/relational_databases_book_download/src_rdb.zip

В случае возникновения вопросов или обнаружения ошибок, опечаток и иных недочётов в книге, пишите: dbb@svyatoslav.biz.

* * *

Если вам понравилась эта книга, обратите внимание на ещё две, написанные в том же стиле:



«Работа с MySQL, MS SQL Server и Oracle в примерах»

В книге: 3 СУБД, 50+ примеров, 130+ задач, 500+ запросов с пояснениями и комментариями. От SELECT * до поиска кратчайшего пути в ориентированном графе; никакой теории, только схемы и код, много кода. Будет полезно тем, кто: когда-то изучал язык SQL, но многое забыл; имеет опыт работы с одним диалектом SQL, но хочет быстро переключиться на другой; хочет в предельно сжатые сроки научиться писать типичные SQL-запросы.

Скачать: http://svyatoslav.biz/database_book/

«Тестирование программного обеспечения. Базовый курс.»

В основу книги положен десятилетний опыт проведения тренингов для тестировщиков, позволивший обобщить типичные для многих начинающих специалистов вопросы, проблемы и сложности. Эта книга будет полезна как тем, кто только начинает заниматься тестированием программного обеспечения, так и опытным специалистам — для систематизации уже имеющихся знаний и организации обучения в своей команде.

Скачать: http://svyatoslav.biz/software_testing_book/

---

262 «Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International». [https://creativecommons.org/licenses/by-nc-sa/4.0/legalcode]

Производственно-практическое издание

Куликов Святослав Святославович

РЕЛЯЦИОННЫЕ БАЗЫ ДАННЫХ В ПРИМЕРАХ

Практическое пособие
для программистов
и тестировщиков

Публикуется в авторской редакции

Дизайн обложки *Р. В. Щуцкий*
Компьютерная вёрстка *В. В. Анцух*

Подписано в печать 07.09.2020.
Формат 60x84/8. Бумага офсетная.
Усл. печ. л. 49,29. Уч.-изд. л. 40,17.
Тираж 1000 экз. Заказ 774.

Издатель и полиграфическое исполнение:
ОДО «Издательство “Четыре четверти”».
Свидетельство о государственной регистрации
издателя, изготовителя и распространителя печатных изданий
№ 1/139 от 08.01.2014, № 3/219 от 21.12.2013.
Ул. Б. Хмельницкого, 8-215, 220013, г. Минск.
Тел./факс: (+375 17) 331 25 42. E-mail: info@4-4.by

Об авторе:

Святослав Куликов

Специалист по подготовке персонала EPAM Systems, кандидат технических наук, доцент Белорусского государственного университета информатики и радиоэлектроники.

Автор и ведущий тренингов “Основы функционального тестирования”, “Автоматизация тестирования”, “Веб-разработка с использованием PHP”.

Более двадцати лет в IT, более десяти лет опыта подготовки тестировщиков и веб-разработчиков.

Блог автора: http://svyatoslav.biz

epam | EPAM training center