МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Физический факультет
Кафедра физико-технической информатики

А.А. Сабитов, С.А. Пирогов

ВВЕДЕНИЕ В СУБД

(учебное пособие)

Новосибирск
2012

Сабитов А. А., Пирогов С.А., Введение в СУБД: учебное пособие // Новосиб. гос. ун-т. Новосибирск, 2012. – 81 с.

Учебное пособие разработано для учебного курса «Введение в СУБД», читаемому бакалаврам физического факультета Новосибирского Государственного университета. В пособии описаны исторические аспекты возникновения современных СУБД, рассмотрены вопросы, связанные с теоретическими основами, архитектурой и использованием реляционных баз данных. Также рассматриваются вопросы, связанные с современными перспективными направлениями в области нереляционных баз данных и хранилищ данных.

Пособие будет, несомненно, интересно и полезно широкому кругу читателей: студентам, изучающим соответствующий курс в рамках образовательной программы, так и научным сотрудникам, желающим ознакомиться с одной из важнейших областей прикладного применения компьютерных систем.

Рецензент к.ф.-м.н., доцент, заведующий кафедрой ФТИ ФФ НГУ И. Б. Логашенко.

Учебное пособие подготовлено в рамках реализации Программы развития НИУ-НГУ на 2009–2018 г.г.

Введение.

Человечество всегда стремилось накопить и сохранить информацию, которая так или иначе оказалась в его распоряжении. Это прослеживается, в том числе, и на примере развития вычислительной техники. В общем виде, ее развитие условно можно разделить на два направления: использование вычислительных машин для проведения различного рода расчетов и построение информационных систем. Второе направление исторически возникло несколько позже первого, но на настоящий момент оно столь же, а возможно и более, важно чем первое.

Собственно, рассмотрение систем хранения информации и методов ее обработки и является целью данного курса.

Значительным шагом к современным системам хранения и обработки данных явилось создание концепции *файловых систем*[1]. Для нашего рассмотрения достаточно сказать, что файловые системы позволяют создавать именованные объекты, называемые *файлами*. При этом данные, хранящиеся в файле могут рассматриваться либо как поток байтов (такой подход характерен для UNIX-подобных систем), либо как последовательный набор записей (например файловая система DEC VMS). С целью упрощения работы с файлами, файловая система позволяет создавать *директории*, которые могут содержать либо вложенные директории, либо файлы. Очевидно, что используя файловую систему можно построить определенную древовидную иерархию из директорий и файлов, такую, что структура дерева будет определенным образом отражать логическую структуру хранимой информации, а файлы, содержащиеся в дереве директорий, будут хранить собственно данные.

Многопользовательские операционные системы требуют обязательного наличия механизма *прав доступа* у используемых ими файловых систем. Используя этот механизм можно определить, какой пользователь имеет право читать или записывать данные в тот или иной файл данных. Тем самым появляется возможность использования данных несколькими пользователями.

Основным недостатком такой схемы является невозможность гарантированного поддержания данных в непротиворечивом состоянии. Для решения этой проблемы естественным шагом является создание программного комплекса, который имеет исключительное право доступа к файлам данных. Основной задачей данного программного комплекса является поддержание данных в непротиворечивом состоянии. Все операции с данными выполняются через интерфейс, предоставляемый данным программным комплексом.

Подобный программный комплекс принято называть *системой управления базой данных*, или сокращенно СУБД, а собственно набор данных — *базой данных*.

Первые СУБД позволяли хранить данные только в виде древесной иерархи, что является следствие естественного перехода от хранения данных на основе файлов к СУБД. Такая структура вполне удобна для хранения данных, отражающих иерархические структуры реального мира, например, структуру фирмы или организации. Базы данных с подобной структурой принято называть *иерархическими базами банных*.

Вполне очевидно, что далеко не всю информацию можно описать в терминах иерархических отношений. Примером этого может служить распределение задач между сотрудниками. В частности, один и тот же проект или задача может выполняться несколькими сотрудниками. В то же самое время один сотрудник может работать над несколькими проектами. Иными словами граф отношений между сотрудниками и проектами является не деревом, а более общим графом — сетью. Для отражения такого рода отношений

1 Подробное рассмотрение понятия файловой системы выходит за рамки данного методического пособия. Соответствующая информация может быть найдена в учебной литературе по организации операционных систем.

в иерархических СУБД требуется создание и поддержание двух параллельных структур для отражения иерархий разделения задач по сотрудникам и выполнения задач данным сотрудником. Что, очевидно, приводит к проблеме поддержания этих структур в согласованном состоянии.

Для разрешения подобной проблемы были разработаны *сетевые* СУБД, позволяющие хранить данные, отношения между которыми описываются сетевым графом.

Следующим шагом в эволюции СУБД было создание реляционной модели, которая будет рассмотрена в главе «Реляционная модель». При использовании СУБД, вообще говоря, легко заметить, что база данных содержит большое число записей, которые можно отнести к относительно небольшому набору категорий. При этом весь набор записей одной и той же категории, можно представить как некоторого рода таблицу, колонками которой являют поля записи, а строками — сами записи. Если рассматривать такие таблицы как множества, то к ним становятся применимы некоторые операции теории множеств. Как будет показано, все операции определенные над такими множествами в результате всегда дают множество такого же рода, что и исходные.

Такой подход позволяет, используя довольно простой математический аппарат, с одной стороны всегда работать с однородными объектами, а с другой — отражать в структуре базы данных произвольно сложные отношения между объектами реального мира. Действительно, взаимосвязи между записями можно отразить в виде дополнительной таблицы, запись которого содержит адреса связываемых записей из разных таблиц. В силу указанных причин в настоящее время такие СУБД получили наибольшее распространение.

Развитие объектно-ориентированных технологий привело к появлению класса объектно-ориентированных СУБД, которые позволяют трактовать таблицы и записи реляционных СУБД, как объекты, имеющие набор свойств и методов.

Реляционная модель.

Первое формальное описание реляционной модели было представлено в работе Е. Кодда "Реляционная модель данных для больших разделяемых банков данных".

Основная идея реляционной модели состоит в том, что все данные представляются в виде таблиц (которые Кодд назвал отношениями), обладающих специальными свойствами.

Согласно Дейту, реляционная модель связана с тремя аспектами данных:

структурным —
: все данные располагаются в отношениях, на расположение данных накладывается ряд требований, которые будут рассмотрены в последующих разделах данной главы и в главе «Проектирование баз данных»;

целостным —
: каким условиям должны удовлетворять данные (эти условия, как правило, реализуют бизнес-правила, определяемые на стадии проектирования базы данных[2]), расположенные в таблице, и взаимосвязи между данными в разных таблицах (такую взаимосвязь принято называть ссылочной целостностью);

манипуляционным —
: определяется инструментарий для работы с данными, включающий в себя два альтернативных подхода: реляционную алгебру и реляционное исчисление.

Общие определения.

Определим основные понятия реляционной модели.

Прежде всего дадим неформальные определения для понятий, используемых в реляционной модели. После чего рассмотрим более формальные определения этих понятий.

Атрибут 1	Атрибут 2	Атрибут 3	Атрибут N
Значение 1 1	*Значение 2 1*	*Значение 3 1*	*Значение N 1*
Значение 1 M	*Значение 2 M*	*Значение 3 M*	*Значение N M*

Любые данные, используемые в программировании имеют тот или иной *тип данных*. Как правило, в программировании рассматриваются типы данных, которые можно отнести к трем категориям:

- простые (или атомарные) типы данных;
- структурированные типы данных (массивы, структуры и т.п.);
- ссылки.

Простые типы данных не обладают внутренней структурой и, по-другому, называются скалярами. Разумеется, атомарность простых типов данных относительна — например, строка, содержащая название организации, состоит из последовательности символов. Однако, важно, что при рассмотрении такой строки как последовательности символов, мы теряем смысл, содержащийся в строке. Т. е. с этой точки зрения, строка неделима. В зависимости от конкретного языка программирования список простых типов данных может определенным образом варьироваться, но все простые типы данных можно отнести к трем основным категориям:

- булевский
- числовой
- строковый

2 Вопросы, связанные с проектирование баз данных, рассматриваются в главе «Проектирование баз данных».

Доменом принято называть множество всех допустимых значений. В отличие от типа данных, понятие домена несет в первую очередь семантическую нагрузку. Домен всегда определяется на простых типах данных или других доменах и, в пределах базы данных, имеет уникальное имя. При своем определении домен получает некое логическое условие, которое позволяет определить, принадлежит ли то или иное значение данному домену[3].

Основное назначение доменов состоит в том, что они ограничивают возможность операций над данными, имеющими одинаковый тип данных, но имеющими различную природу. Например, вес и длинна, в качестве допустимых значений, имеют все положительные вещественные числа, тем не менее, они принадлежат к различным доменам.

Атрибутом отношения называют пару <A:D>, где A – имя атрибута, а D – домен, на котором определяется атрибут. Имя атрибута должно быть уникально в пределах отношения и может совпадать с именем домена.

Отношение, определенное на множестве доменов (D_1, ... D_n) представляет из себя сущность, состоящую из двух частей: тела и заголовка. *Заголовок отношения* содержит множество атрибутов (<A_1:D_1>, ... <A_n:D_n>). В свою очередь *тело отношения* содержит множество *кортежей*, каждый из которых имеет следующий вид: (<A_1:V_1>,...<A_n:V_n>), где A_i — имя атрибута, а V_i — его значение, принадлежащее домену D_i.

Количество кортежей в отношении называют *кардинальным числом* или *мощностью* отношения. Количество атрибутов — *степенью* или *арностью* отношения.

Множество атрибутов K отношения R принято называть *потенциальным ключём*, если:

- атрибуты из K однозначно идентифицируют кортежи в R, и
- при удалении любого атрибута из K, множество K перестает однозначно идентифицировать кортежи R.

Очевидно, что отношение может иметь несколько потенциальных ключей. (Допустим, что отношение описывает автомобили, принадлежащие предприятию. Тогда, потенциальным ключём могут выступать: государственный регистрационный номер и номер шасси.) При построении конкретной базы данных один из потенциальных ключей выбирается в качестве *первичного ключа*.

Выбор первичного ключа определяется спецификой приложения и будет рассматриваться в главе «Проектирование баз данных».

Свойства отношений.

Перечислим фундаментальные свойства отношений, которые следуют из данных выше определений.

Уникальность кортежей.

Любое отношение в любой момент времени содержит только уникальные кортежи. Иными словами, в отношении нет двух одинаковых кортежей. Это свойство является прямым следствием того, что отношение определяется как множество кортежей[4].

На это свойство можно посмотреть и несколько с другой стороны. Можно себе представить, что домены, лежащие в основе отношения, образуют некое многомерное пространство, а каждый конкретный кортеж задает точку в этом пространстве. Тогда становится очевидным, что каждый кортеж должен быть уникален, т. к. в противном случае нет никакой возможности однозначно адресовать конкретный кортеж из нескольких дубликатов.

---

3 Вообще говоря, не всегда можно задать такое формальное условие. Например, нет возможности указать, какая строка является допустимым названием корабля, а какая нет. (Сравните "Луч" и "Зщшгне".)

4 В классической теории множеств все элементы множества различны.

Атомарность значений атрибутов.

Данное свойство следует из того факта, что все значения атрибута принадлежат определенному домену, который, в свою очередь, основывается на простом типе данных.

Неупорядоченность кортежей.

Неупорядоченность кортежей также является следствием определения отношения как множества кортежей. На практике это позволяет СУБД использовать гибкие схемы хранения данных и доступа к ним.

Существующие СУБД позволяют сформировать запрос, который вернет данные, упорядоченные по одному или нескольким атрибутам, но такой *результат* не является *отношением*. Более того, при сохранении результатов запроса в базе данных, **нет** никакой **гарантии**, что упорядоченность данных сохранится.

Неупорядоченность атрибутов.

Аналогично предыдущим двум пунктам, неупорядоченность атрибутов следует из определения отношения, а точнее говоря, из определения заголовка отношения. С теоретической точки зрения, это позволяет модифицировать отношение, как путем добавления или переопределения атрибутов, так и путем их удаления. На практике это никогда не выполняется и за порядок атрибутов принимается порядок их следования при создании отношения.

Реляционная алгебра и реляционное исчисление.

Как уже было сказано в начале этой главы, реляционная модель содержит описание двух подходов к вопросу манипулирования данными, хранящимися в отношениях. Первый из них принято называть *реляционной алгеброй*, второй — *реляционным исчислением*.

Реляционная алгебра построена на операциях над множествами и определяет результирующее отношение как набор исходных отношений и, примененной к ним, последовательности операций.

Реляционное исчисление основано на теории предикатов и описывает результирующее отношение, как набор предикатов первого порядка, которым оно должно удовлетворять.

Оба эти подхода эквиваленты между собой, т. е. любой результат, представимый посредством реляционной алгебры, может быть представлен средствами реляционного исчисления и наоборот.

Реляционная алгебра.

Поскольку реляционная модель основана на теории множеств, к отношениям применимы четыре основные операции, определенные над обычными множествами:

объединение —
: результирующее отношение содержит все кортежи, принадлежащие любому из аргументов

пересечение —
: результирующее отношение содержит все кортежи, принадлежащие одновременно всем аргументам

разность —
: результирующее отношение содержит все кортежи первого аргумента, которые не принадлежат второму аргументу

Декартово (или прямое) произведение —

результирующее отношение содержит кортежи, которые являются конкатенацией кортежей аргументов, т. е. к каждому кортежу первого аргумента, дописываются все кортежи второго аргумента

Кроме указанных операций, реляционная алгебра содержит набор специальных операций:

проекция —

результирующее отношение содержит кортежи исходного отношения, в которых оставлены только указанные атрибуты

выборка —

результирующее отношение содержит кортежи исходного отношения, которые удовлетворяют некоторому условию

соединение —

результирующее отношение содержит конкатенацию кортежей аргументов, которые удовлетворяют условию соединения

деление —

если отношение A определено на множестве атрибутов ($c_1,...c_n,b_1,...b_m$), а отношение B — на множестве атрибутов ($b_1,...b_m$), то частным от деления A на B будет отношение C, определенное на множестве ($c_1,...c_n$), такое, что для любого ($c_1,...c_n$) из C и всех ($b_1,...b_m$) из B, в A найдутся кортежи ($c_1,...c_n,b_1,...b_m$)

При внимательном рассмотрении, заметно, что некоторые операции из вышеперечисленных можно выразить через другие операции. Так, например, операцию соединения можно представить через операции прямого произведения и выборки, операцию пересечения можно выразить через операцию разности, а операцию деления можно выразить через операции прямого произведения, проекции и разности. Прочие операции являются примитивными, в том смысле, что они не могут быть выражены через набор других операций.

Вообще говоря, принято выделять несколько типов операции соединения. Наиболее общий из них --- *тэта-соединение*. Его название объясняется тем, что любой из операторов сравнения $=, \neq, \leq, \geq, <, >$ в условии соединения, в общем случае, принято обозначать символом Θ. В силу того, что на практике чрезвычайно часто на месте Θ используется знак равенства, его принято называть *естественным* или *экви-соединением*, по этой же причине существует масса хорошо проработанных методов оптимизации таких запросов.

В реляционной алгебре существуют еще две операции, которые формально не являются операциями над множествами, но необходимы для нормальной работы с реляционной базой данных:

переименование —

переименовывает атрибуты исходного отношения и возвращает все его кортежи

присваивание —

сохраняет вычисленный результат в указанном отношении

Реляционное исчисление.

Базовыми понятиями реляционного исчисления являются: понятие *кортежной переменной*, *области определения* кортежной переменной и *правильно построенной формулы*.

В зависимости от того, что является областью определения кортежной переменной различают *исчисление доменов* и *исчисление кортежей*. В первом случае, областью определения переменной является домен, иными словами, переменная может принимать любое значение, принадлежащее данному домену. Во-втором случае, областью определения

переменной является отношение, т. е. переменная может принимать значение некоторого кортежа из данного отношения.

Будем обозначать кортежную переменную следующим образом: RANGE OF T IS X_1, X_2,..., где T — имя кортежной переменной, X_i — имена отношений, либо выражение исчисления кортежей. Если указывается более одного элемента в списке, то заголовки отношений, указанных в списке, должны совпадать. Областью определения переменной при этом становится объединение кортежей всех отношений.

Выражение исчисления кортежей принято записывать в следующей форме: (v_1,...v_n) [WHERE wff], где (v_1,...v_n) — *список целевых элементов*, а wff — правильно построенная формула. Квадратные скобки в данном случае (и ниже в этом разделе) означают, что элементы, в них заключенные, могут быть опущены.

Любой целевой элемент является либо просто именем переменной, либо конструкцией вида T.a [AS na], где T — имя кортежной переменной, a — имя атрибута отношения, сопоставленного с кортежной переменной, na — новое имя переменной. Новое имя **должно** быть указанно, если список целевых переменных содержит повторяющиеся имена.

Правильно построенная формула (wff) может содержать:

- *условие* (cond)
- NOT wff
- cond AND wff
- cond OR wff
- IF cond THEN wff
- EXISTS var (wff)
- FORALL var (wff)

Подразумевается, что условие может содержать либо скалярное сравнение, содержащее скалярные константы и ссылки на атрибуты в виде T.a, либо правильно построенную формулу, заключенную в круглые скобки.

Квантор EXISTS требует, что бы на всем наборе значений переменной var, существовало хотя бы одно значение, для которого (wff) принимает истинное значение. Квантор FORALL, напротив, требует, что бы (wff) принимало истинное значение для всех значений из набора.

Для получения значений, описываемых выражением исчисления, используется оператор RETRIVE применительно к целевому набору переменных.

Предположим, что у нас имеются два отношения:

A	
NUM	**STR_A**
1	aaa
2	asd
3	qwe
7	zaq

B	
NUM	**STR_B**
2	aaa
2	aaa
4	main
5	blue

отношения для примеров

Рассмотрим на примере этих отношений, как используется реляционное исчисление.

1. Получить все пары чисел, которым соответствуют совпадающие строки.

```
RANGE OF vA IS A;
RANGE OF vB IS B;
RETRIVE (vA.NUM AS aNUM, vB.NUM AS bNUM) WHERE vA.STR_A = vB.STR_B;
```

2. Получить кортежи отношения A, с номером больше 2

```
RANGE OF vA IS A;
RETRIVE (vA) WHERE vA.NUM > 2;
```

SQL.

История появления. Стандартизация.

SQL (Structured Query Language, «язык структурированных запросов») — язык программирования, применяемый для манипулирования данными в реляционных базах данных. В настоящее время он является стандартом для работы с реляционными СУБД.

История появления SQL уходит в далекие 1970-ые годы, когда компания IBM разработала специальный язык SEQUEL для своей экспериментальной реляционной СУБД IBM System R. SEQUEL — это аббревиатура от Structured English QUEry Language. В дальнейшем SEQUEL был переименован в SQL. Разработка SEQUEL не велась с чистого листа, в его основу легли все достоинства реляционной модели, подкрепленной математическим аппаратом реляционной алгебры и реляционного исчисления. При этом простой синтаксис и сравнительно небольшое количество операндов сделали язык легким в использовании, а также позволили достаточно быстро набрал популярность.

К 1980-м годам существовало несколько коммерческих независимых решений реляционных СУБД, каждый из которых использовал собственную реализацию языка запросов. Данный факт усложнял процесс переносимости программного обеспечения между различными СУБД и тормозил развитие языка в целом. Поэтому не удивительно, что уже в 1983 Международная организация по стандартизации (ISO) и Американский национальный институт стандартов (ANSI) предприняли попытку разработать некий единый стандарт языка запросов. Таким образом в 1986 году появился первый стандарт SQL-86 (SQL1), в основу которого был взят SQL IBM System R. Теперь производители СУБД имели некий ориентир и следовали ему. Это конечно не избавило от появления различных диалектов SQL, но по крайней мере сделало эти диалекты достаточно близкими друг к другу.

Следующая версия стандарта SQL-89, которая учитывала все не недочеты первой версии и новые пожелания, появилась уже через три года.

Далее последовали SQL-92[5], SQL:1999[6], SQL:2003[7], SQL:2006, SQL:2008, SQL:2011. Стандартизация языка дала мощный толчок в использовании СУБД прикладными программами, и обеспечило высокую степень их независимости от конкретного типа используемой СУБД. Что является немаловажным фактором при использовании коммерческих и некоммерческих СУБД.

Как уже было сказано не смотря на все стандарты, каждая реализация СУБД имеет свой собственный диалект SQL, зачастую не вполне совместимый с другими диалектами. Можно сказать, что все попытки стандартизации закончились на выделении основной части стандарта в раздел «SQL/Foundation». А вопрос совместимости СУБД, сводится к вопросу поддержки ими этой основной части. Поддержка остальных возможностей оставляется на усмотрение производителя конкретной СУБД. Это означает, что на данный момент не существует полной гарантии, что ваше программное обеспечение может быть легко перенесено с одной СУБД на другую нет.

Как правило, проблема переносимости на другую СУБД при разработке программного обеспечения решается выделением некоторого интерфейсного уровня для работы с базой данных. Все запросы к базе данных делаются через этот интерфейс, а конкретная реализация данного интерфейса уже ориентируется под определенную СУБД. Таким образом для перехода на другую СУБД вам достаточно сделать новую реализацию интерфейсного уровня, при этом изменений кода самого программного продукта не происходит. Сложность реализации интерфейсного, если использование SQL остается в рамках «SQL/Foundation», зачастую сво-

5 Ещё известен как SQL2
6 Ещё известен как SQL3
7 Считается, что именно этот стандарт действует в настоящее время

дится к формальности.

С одной стороны, существование множества SQL диалектов только осложняют жизнь простым программистам, ухудшает совместимость программных систем, но, с другой стороны, является своеобразной двигающей силой в развитии языка. Со временем полезные новшества заимствуются между диалектами, обкатываются и вносятся в новые редакции стандарта.

Благодаря тому, что SQL является высокоуровневым языком программирования, понимание общей концепции языка и знание одного из диалектов, позволяет программисту с лёгкостью перейти при необходимости на другой диалект.

Структура стандарта.

Начиная с SQL1 структура стандарта в значительной изменилась. В 1993 году комитетами ANSI и ISO было решено разделить стандарт на несколько основных глав. Данное разделение было представлено в стандарте SQL99. Структура SQL2003 с небольшими изменениями сохранилась со стандарта SQL99 и дополняется только главой SQL/XML:

- ***Framework (SQL/Framework)*** – Описание общих определений, концепций и требований соответствий стандарту.
- ***Foundation (SQL/Foundation)*** – основная часть, описание функционального ядра данного стандарта. Самая массивная (~800 станиц) и важная глава. Содержит описание всех разделов стандарта.
- ***Call-Level Interface (SQL/CLI)*** – Описание интерфейса уровня вызовов, который используется для динамического запуска SQL инструкций из внешних приложений.
- ***Persistent Stored Modules (SQL/PSM)*** – Содержит описание конструкций процедурных языков дополняющих SQL.
- ***Management of External Data (SQL/MED)*** – Описание SQL-расширения, предназначенного для управления внешними данными.
- ***Object Language Bindings (SQL/OLB)*** – Описание SQLJ, являющегося подмножеством SQL, который позволяет встраивать SQL инструкции в Java.
- ***Information and Definition Schemas (SQL/Schemata)*** – Описание схем информации и определения, представляющих набор инструкций, которые позволяют базе данных описывать саму себя. Все инструкции хранятся в специальной схеме INFORMATION_SCHEMA.
- **SQL Routines and Types Using the Java Programming Language (SQL/JRT)** – Описание расширение стандарта SQL, позволяющего вызывать статические Java-методы как подпрограммы из SQL-приложений и использовать Java-классы как пользовательские SQL-типы.
- **XML-Related Specifications (SQL/XML)** – Описание SQL-расширения для использования XML в сочетании с SQL.

Диалекты SQL.

Как уже было отмечено выше каждый производитель разрабатывает свой собственный язык запросов. Стандартизация SQL способствовала стремлению производителей приводить свои диалекты как можно ближе стандарту, но не привела к единому языку, который бы использовался всеми. Каждый производитель в первую очередь преследует интересы своих пользователей, привнося в свой диалект новшества требуемые своему сообществу. Такой путь развития с одной стороны отдаляет диалекты друг от друга, но с другой служит дополнительным двигателем развития. Наиболее удачные решения заимствуются другими

диалектами при этом привнося свои собственные идеи. Поэтому не удивительно, что многие новшества впервые появляются в диалектах, а уже затем входят в стандарт. Специальных названий для диалектов SQL нет. Обычно, когда хотят подчеркнуть, что имеется ввиду как-то конкретный диалект, то говорят *диалект название СУБД*. Например, диалект Oracle или диалект MySQL. А вот названия процедурных расширений имеют свои собственные названия. Ниже перечислены процедурные расширения наиболее популярных диалектов SQL:

- **PL/SQL** – Procedural Language/SQL. Используется в СУБД Oracle.
- **Transact-SOL** – Используется в СУБД MS SQL Server и Sybase Adaptive Server. Оба производителя используют общую платформу, но с самого начала выбрали разные пути развития.
- **PL/pgSQL** - Procedural Language/PostgreSQL Structured Query Language. Используется в СУБД PostgreSQL.
- **SQL/PSM** - SQL/Persistent Stored Module. Используется в СУБД MySQL.
- **PSQL** - Procedural SQL. Используется в СУБД InterBase и FirebirdSQL.
- **SQL PL** - SQL Procedural Language. Используется в СУБД IBM DB2. Один из новых диалектов, появившийся после стандарта, поэтому наиболее совместим с ним.

Основные отличия SQL. Преимущества и недостатки.

В отличие от традиционных языков программирования, таких как Java или C, язык SQL описывает желаемый результат, но не описывает алгоритм получения этого результата, оставляя последний на усмотрение СУБД. SQL относится к группе информационно-логических языков. Основными функциями языка является описание, извлечение и изменение данных.

Преимущества SQL:

1. *Легкость в изучении.*
2. *Платформ-независимость.*
3. *Наличие стандарта и относительная легкость миграции между СУБД.*
4. *Многие программные продукты, использующие базы данных, поддерживают работу с несколькими СУБД и предоставляют пользователям выбор.*
5. *Декларативность.*
6. *Разработчик программного обеспечения говорит СУБД, что он хочет сделать, а СУБД решает как это сделать.*

Недостатки SQL:

1. *Отклонение от реляционный модели.*
   Не смотря на то, SQL называют реляционным языком в чистом виде таковым он не является. SQL имеет ряд существенных отклонений от реляционной модели предложенной Эдгаром Кодом:
   1. Отсутствие понятия домена.
   2. Явное указание порядка колонок (слева на право).
   3. Использование указателей.
   4. Механизм неопределённых значений (null).
   5. Наличие дубликатов.
   6. Избыточностью

   и др.
2. *Сложность формулировок и громоздкость.*
   С момента своего появления стандарт впитал в себя такое огромное количество различных усовершенствований и нововведений, что порой конструкции языка выглядят нечитабельными. Справедливости ради нужно отметить, что базовая часть языка не

изменилась и продолжает поддерживаться в первоначальном виде.

3. *Наличие большого числа диалектов и отступления ими от стандарта SQL,*

Структура реляционной базы данных и ключевые понятия.

Очевидно, что информацию, описывающую произвольный набор отношений, можно хранить в фиксированном наборе служебных отношений. Причём последний будет описывать любое количество произвольно сложных отношений, расположенных в базе данных. Например, он должен содержать отношение, описывающее какие атрибуты содержатся в том или ином отношении БД, или отношение, описывающее на каком домене определён тот или иной атрибут. Такой набор служебных отношений принято называть *словарём базы данных*. Таким образом, словарь базы данных содержит мета-информацию о структуре конкретной базы данных и необходим для нормальной работы самой СУБД. Информация о структуре самого словаря, хотя и может храниться в самом словаре, считается предопределённой и зависит от версии СУБД.

Таблица.

Основным объектом, с которым имеет дело реляционная СУБД, является *таблица* (table) — очень близкий аналог отношения реляционной модели. Но не смотря на сильную схожесть, таблица РСУБД имеет существенные отличия от отношения. Первое из них — атрибут таблицы может содержать неопределённое значение, которое обозначается как *NULL*, что недопустимо в реляционной модели. Применительно к атрибутам реляционных СУБД принято применять термины *поле*, *колонка* или *столбец*. Аналогично, кортеж принято называть *строкой* или *записью*. Одним из существенных отличий от классической реляционной модели является возможность хранения нескольких одинаковых строк в таблице. Более того, атрибуты таблицы, в отличие от атрибутов отношения, упорядочены. То есть явное указание порядка колонок с лева на право.

Представление.

Наравне с таблицей ещё одним из ключевых объектов РСУБД является *представление* (view), которое можно представить как хранимый запрос к базе данных, выглядящий для пользователя как таблица. Представление не хранит реальных данных. Содержание представления формируется каждые раз, когда представление используется в SQL запросе. Часто представление называют виртуальной или логической таблицей, представляющей *алиас* к запросу. Так как представление вычисляется динамически, то изменение данных в реальных таблицах, немедленно влияет на содержание представлений, основанных на этих таблицах.

Так как содержание предоставления — это не что иное, как результат SQL запроса, то способы и возможности построения представления ограничиваются только возможностями SQL диалекта, с которым вы работаете. Таким образом представлением может является: выборка части таблицы, объединение нескольких таблиц, результат обработки данных (например арифметическая сумма столбцов или конкатенация строк) и т.п.

Использование представлений является очень удобным в прикладных программах. Представления помогают скрыть сложность организации данных и запросов к ним, позволяют отделить работу с данными от схемы их хранения. Со стороны прикладной программы структура данных представлена представлениями. На самом же деле организация хранения данных может выглядеть совершенно иначе. Такое разделение позволяет независимо модифицировать схему хранения данных и саму прикладную программу. В первом случае мы имеем дело с изменением таблиц, во втором случае нам достаточно модифицировать только представления. Благодаря такому разделению уровень

представлений может являться дополнительным уровнем защиты данных. Пользователь получает права доступа только на те данные таблиц, которые представлены представлением. Другие данные, находящиеся в тех же таблицах, но не затрагивающиеся этим представлением остаются не доступными.

С одной стороны удобство использования представлений может обратиться в увеличение время получения данных. Особенно в сложных системах. Это происходит из-за того, что запрос с представлением обрабатывается СУБД точно так же, как и обычный запрос, подставляя вместо представления соответствующий подзапрос. Современные СУБД с развитыми возможностями оптимизации, перед выполнением запроса с представлением, могут проводить оптимизацию запроса верхнего уровня и запроса, определяющего представление, с целью минимизации затрат на выборку данных. Также стоит отметит, что в силу того, что SQL-запрос представления, зафиксирован на момент его создания, СУБД получает возможность применить к этому запросу оптимизацию или предварительную компиляцию, что положительно сказывается на скорости обращения к представлению, по сравнению с прямым выполнением того же запроса из прикладной программы.

Индекс.

Для ускорения работы запросов, которые выбирают информацию из таблицы по некоторому заданному условию, очень часто создаются *индексы*. Индекс — это объект СУБД, который содержит информацию о расположении строк таблицы на физическом носителе в зависимости от значений атрибутов, на которых он построен. Поддержание индекса в актуальном состоянии требует от СУБД дополнительных накладных расходов, при внесении в таблицу новых данных или изменении существующих. Использование или не использование индекса определяется оптимизатором СУБД в момент выполнения запроса, и является прозрачным для программиста или конечного пользователя.

Одним из самых популярных подходов при организации индексов является создание различного рода модификаций классического B-дерева. В число часто вводимых модификаций B-дерева входит упреждающее расщепление листовой страницы дерева, переливания данных между соседними страницами и т. п. При создании многопользовательской СУБД также создаются различные механизмы для совместного доступа к индексу. Такие модификации B-дерева в литературе принято называть B+ или B*-деревом. Необходимо иметь ввиду, что вносимые модификации привносят с собой и возможность разбалансирования дерева.

Еще одним способом организации индекса является создание хэша.

В зависимости от СУБД, перечисленные объекты могут либо прямо отображаться на файлы операционной системы (так делает, например, MySQL), либо храниться в специальном логическом хранилище — *табличном пространстве*, — которое может содержать один или несколько файлов операционной системы, либо располагаться на "сыром" устройстве для повышения производительности системы (так делает, к примеру, Oracle).

Хранимые процедуры и функции.

Хранимые процедуры и функции являются объектами базы данных, содержащими набор SQL инструкций. Пожалуй главный смысл использования хранимых процедур и функций заключается в том, что мы можем с их помощью выделить и оформить некоторый повторяющейся кода, который может быть переиспользован в дальнейшем. Здесь можно провести параллель с языками высокого уровня:

- могут быть входные/выходные параметры и локальные переменные, над которыми могут совершаться числовые вычисления и символьные операции, выполняться операции присваивания и сравнения;

- возможны циклы и ветвления;
- вызов других процедур и функций с использованием их результата;

Кроме того в хранимых процедурах могут использоваться стандартные DDL и DML операции, а в функциях только оператор SELECT (различия между хранимыми процедурами и функциями описаны ниже). Различные современные СУБД позволяют использовать в хранимых процедурах и функциях языки высокого уровня: C/C++, Java, Perl, Python и др., что в значительной степени расширяет возможности и увеличивает круг задач, которые при этом могут быть решены.

Понятие хранимой процедуры и функции достаточно близки друг другу, но они совершенно по разному используются. Часто главным отличием функции от процедуры называют возможность возвращать значение - это несколько не так. Давайте рассмотрим основные хранимых процедуры и функции:

- В отличие от функции, процедура всегда вызывается посредством функции CALL или EXECUTE, что делает не возможным использование процедуры в SQL-запросе.

```
CALL procedure_name(parameters_list)
   OR
EXECUTE procedure_name(parameters_list)


CALL function_name(parameters_list)
   OR
EXECUTE function_name(parameters_list)
   OR
SELECT function_name(column_name) from table_name
```

- Функция не может модифицировать данные в базе данных, она может совершать над базой данных только операции выборки.
- В функции не допускаются объявление глобальных курсоров, открывать и закрывать транзакции. Также имеются ограничения на использование некоторые системные функции.
- Возвращаемое значение. Функция всегда должна иметь возвращаемое значение. Также стоит отметить, что некоторые диалекты SQL допускают в качестве возвращаемых значениях кроме *скалярных* (простых типов) данных также и *табличные* (результат выполнения такой функции является табличная переменная — представление). К *табличной функции* позволяется производить SQL запрос.

```
SELECT * from function_name(parameters_list)
```

Материализованное представление.

Материализованное (или материальное) представление впервые появилось в СУБД Oracle. Отличие от обычного представления заключается в том, что материализованное представление использует заранее вычисленные итоговые данные и результаты соединений таблиц (своего рода кеш), что значительно уменьшает время выполнения запросов. Для того, чтобы представление находилось в актуальном состоянии используются триггеры или периодические синхронизации.

Триггер.

Часто триггером называют хранимую процедуру особого типа, главным отличием которой является то, что она не вызывается пользователем. Триггер вызывает сервером автоматически при попытке изменения данных в таблицах, с которыми данный он связан. Триггер — это механизм, используемый для поддержания целостности данных, с помощью которого можно реализовать сложные проверки или последовательное изменение зависимых данных. Все изменения в рамках триггера рассматриваются как одна транзакция. В случае возникновения исключения или нарушения целостности данных данная транзакция

откатывается, что приводит к возврату всех сделанных изменений.

Использование триггера является ресурс затратной операцией, поэтому стоит избегать использования триггера, если можно обойтись обычной хранимой процедурой. Также нужно учитывать, что неправильная организация логики триггера может привести к потере данных или к бесконечной рекурсии (как правило все современные СУБД поддерживают максимальную вложенность, но это лишь искусственное ограничение), поэтому при разработке нужно быть предельно аккуратным и тщательно проводить тестирование.

Помимо таблиц триггеры могут создаваться и для представлений. Момента запуска триггера определяется задается в момент создания при помощи ключевых слов before или after, соответственно триггер будет активирован до или после наступления события.

Различные реализации СУБД поддерживают разный набор операторов, которые могут быть использованы в триггере.

Последовательность.

Последовательность — это объект СУБД, предназначенный для организации натурального ряда чисел. Последовательность имеет ряд параметров, которые задаются при ее создании: шаг; начальное, минимальное и максимальное значение; зацикливание и др. В тех случаях, когда первичный ключ не несет определенного смысла, а служит лишь для обеспечения уникальности, последовательность идеально подходит в качестве первичного ключа.

Синонима.

Синоним — это объект СУБД, являющимся альтернативным именем (*алиасом*) для другого объекта СУБД, который может располагаться как на локальном, так и на удаленном объекте. Синоним предоставляет уровень абстракции, с помощью которого клиентская часть отвязаться от изменений в наименовании базовых объектах. Синонимы особенно востребованы в иерархически сложных и переиспользуемых системах. Например, есть база данных *Data1* с таблицей *Table1*, расположенной на сервере *Server1*. Полный путь до такого объекта: *Server1.Table1.Data1*. Понятно, если мы работаем с полным именем, то изменение имени любой его части потребует модификации клинического приложения. При использовании синонима на *Server1.Table1.Data1*, достаточно будет только пересоздать синоним.

Домен.

Домен — это допустимое подмножество значений данного типа. Не путайте под доменом множество значений типа данных. Например, домен *русский словарь* определятся на русском алфавите, и представляет набор всех слов на русском языке. Помимо ограничения, что русские слова содержат только буквы алфавита, есть правила правописания. Не каждый набор русских букв может являться словом.

Подавляющее большинство СУБД в настоящий момент не имеют понятия домена, что является важным отличием между реляционной моделью и её реализацией в виде СУБД[8].

Структура SQL.

Язык SQL очень объёмен — так, например, диалект SQL, используемый в СУБД Oracle v.7.3 содержит более 600 страниц и описывает свыше 110 различных команд. Разумеется, в рамках данной главы нет никакой возможности описать этот язык в полном объёме, поэтому рассматривать её надо только лишь как поверхностный обзор языка. При

8 Необходимо отличать понятие домена, применяемое в реляционной модели, и то, что называется доменом в некоторых СУБД (например, PostgreSQL). Последнее является ничем иным, как типом данных с заданными на момент создания ограничениями.

этом целью авторов было сделать некую «выжимку», которая будет легка в понимании и даст необходимый языковой базис.

По типу работы с данными операторы SQL можно разделить на несколько групп. Часто говоря, что язык SQL делится на несколько подъязыков:

DDL —

(Data Definition Language) содержит все команды, которые работают со словарем базы данных. Как правило, сюда входят все команды, начинающиеся со слов: CREATE, ALTER, DROP.

DML —

(Data Manipulation Language) содержит команды, манипулирующие с данными в таблицах. Сюда входят команды: SELECT, INSERT, DELETE, UPDATE.

DCL —

(*Data Control Language*) содержит команды, определяющие доступ к данным: GRANT, REVOKE, DENY.

TCL —

Session Control commands (ALTER SESSION, SET ROLE), *System (or Security) Control Commands* (ALTER SYSTEM), *Embedded SQL Commands* (CLOSE, CONNECT, EXECUTE, FETCH, OPEN) и т.п.

Типы данных в SQL.

В соответствии со стандартом SQL3 определяются три категории типов данных: предопределённые (или встроенные), составные и определяемые пользователем. Каждый тип данных имеет имя. Имена для предопределенных и составных типов являются зарезервированными и описаны в стандарте. Различные диалекты SQL имеют наборы типов, которые специфичны для каждого диалекта, но совпадающие по семантике, что делает изучение и работу с типами легко воспринимаемой. Поэтому при миграции с одного диалекта на другой, вопросов, связанных с типами данных, как правило, не возникают.

В целом, как в стандарте, так и в различных реализациях языка выделяют несколько категорий для предопределенных типов данных:

- **NUMERIC (числовые) —** Целочисленные и вещественные типы данных. Размер данных и точность для вещественных типов могут задаваться.
- **CHARACTER (символьные/строковые типы) —** Строковые типы данных позволяют хранить любую комбинацию символов из допустимого набора. Для некоторых символьных типов можно выставлять ограничение по длине.
- **BINARY STRING (двоичные)** — Двоичные типы данных хранятся как двоичные строковые значения в шестнадцатеричном формате.
- **BOOLEAN (булевы)** — Данное — категория, значение этого типа может принимать только или true(истина) или false (ложь).
- **DATETIME (временные)** — К этой категории относятся все типы данных предназначенные для хранения даты и времени.
- **INTERVAL (интервал)** — Это тоже временной тип, но в отличие группы DATETIME, предназначен для хранения временных интервалов.
- **XML** — Хранение данных в формате XML.

К составным типам данных относят:

- **REF (ссылка)** — Данное — связь, задает ссылку на объект, хранящимся вне базы данной. Например, файл на файловой системе или интернет адрес ресурса.
- **ROW** — последовательность пар (имя, значение).
- **ARRAY** — отсортированная коллекция элементов, имеющая фиксированную длину.

- **MULTISET** — неотсортированная коллекция элементов переменной длины. Данный тип появился в SQL2003.

Давайте в качестве примера рассмотрим типы данных различных СУБД для хранения текстовой информации. В стандарте SQL3 для этого определены следующие типы: CHARACTER— для хранения строк фиксированной длинны, CHARACTER VARYING— для хранения строк переменной длины и CHARACTER LARGE OBJECT — для хранения больших объемов текстовой информации. Для хранения аналогичных объектов в СУБД MySQL используются типы CHAR, VARCHAR и TEXT, соответственно. Для этих же типов данных Oracle использует CHAR, VARCHAR2/VARCHAR и LONG/CLOB.

На этом примере видно, что и MySQL, и Oracle содержат семантически идентичные типы данных, различающиеся между собой именами. Это накладывает ограничения на переносимость прикладных программ между различными СУБД. Однако, необходимо учитывать, что различия могут быть более глубокими, чем просто различные имена типов. Например, тип VARCHAR MySQL требует спецификации размера поля в байтах, который означает некий **характерный** размер строк, которые будут содержаться в данном столбце. И конкретная строка может иметь длину, превышающую указанный размер, за счёт потери производительности системы. Тип данных с тем же именем в Oracle требует указания **максимального** размера строк в байтах, хранящихся в столбе. Соответственно, строка не может превышать указанную величину.

При переносе приложения с одной СУБД на другую все эти различия необходимо учитывать[9].

Ограничения.

Ограничения — это один из важнейших механизмов, ставший уже неотъемлемой частью любой СУБД, обеспечивающий целостность данных. Ограничения — это своего рода набор, задаваемых пользователем, правил, которые определяют является ли данное значение атрибута допустимым или нет. Ограничения могут влиять (блокировать) на работу операций, которые модифицируют данные: INSERT, UPDATE, DELETE. Так, если при транзакции, модифицирующей данные, происходит нарушение ограничений, то данная транзакция отменяется и состояние базы данных возвращается к состоянию до транзакции. Разделяют ограничения по области применения:

(column_constraint) ограничения на уровне слобцов — такие ограничения задаются в момент создания столбца или модификации его параметров и применимы только к нему;

(table_constraint**) ограничения на уровне таблицы** — табличные ограничения используются, если вам необходимо задать ограничения для одного или нескольких столбцам,

Разные диалекты поддерживают разное ограничения, на данный момент можно выделить следующие ограничения, описывающиеся в стандарт SQL :

NOT NULL CONSTRAINT — ограничение на уровне столбца, не позволяет принимать значение NULL.

CHECK CONSTRAINT — проверочные ограничения значений столбца. Позволяет задать ряд условий для области значений столбца. Условия могут быть составными — соединенными логическими операторами AND и OR. В условиях могут использоваться только булевы операторы: <>, =, <, >, <=, >=. Стандарт SQL2003 допускает использование всех предикатов (IN, LIKE и др.)

Пример,

```
CREATE TABLE users
```

9 Справедливости ради, необходимо отметить, что многие СУБД поддерживают типы данных из стандарта, переотображая их на наиболее близкие "родные типы".

```
(id int NOT NULL PRIMARY KEY,
 name VARCHAR(50),
 department_code VARCHAR(20),
 CONSTRAINT v_depart CHECK ( department_ code LIKE '[A-F][0-9][0-9][0-9]'));
```

PRIMARY KEY CONSTRAINT — ограничение на уровне столбца или таблицы позволяет объявить один или несколько столбцов в качестве primary key. Данной ограничение считается ограничением на уровне таблицы в случае, когда primary key состоит из нескольких столбцов, т. е. является составным. Ограничение primary key означает гарантирует, что для каждого значения primary key найдется одна и только комбинации запись в таблице. В силу уникальности можно говорить, что такое ограничение является частным случаем ограничения unique. На ограничение данного типа накладывается ряд правил: не допускается несколько primary key, исключается значение NULL; запрещается накладывать данное ограничение на некоторые типы данных, такие как BLOB, ARRAY, REF, и др.

Примеры ограничения primary key:

на уровне столбца	**на уровне таблицы**
CREATE TABLE users (id int NOT NULL PRIMARY KEY, name VARCHAR(50), family VARCHAR(100), passport VARCHAR(20));	CREATE TABLE users (name VARCHAR(50), family VARCHAR(100), passport VARCHAR(20), CONSTRAINT prkey PRIMARY KEY (name, family));

UNIQUE CONSTRAINT — также, как и ограничение primary key, данное ограничение может быть определено как на уровне столбца, так и на уровне таблицы. Это ограничение гарантирует уникальность значений в одном столбце или в комбинации столбцов. Ограничение unique является потенциальным первичным ключом и на него накладываются все правила, определенные для первичных ключей, кроме того условия, что unique может содержать значения NULL.

REFERENCES CONSTRAINT — ограничение на различного рода (не обязательно полного) совпадений значений столбцов одной таблицы с указанными столбцами другой таблицы. Единственное правило для данного ограничения заключается в совпадении типов связываемых столбцов. Ограничение references позволяет организовать взаимоотношения между таблицами.

FOREIGN KEY CONSTRAINT — ограничение по внешнему ключу. Ограничение foreign key аналогично ограничению references. Оно гарантирует полное совпадение значений связываемых столбцов, обеспечивая так называемую ссылочную целостность. Ограничение по внешнему ключу может ссылаться или на столбец (столбцы) с ограничением unique или primary key.

DML.

Как уже говорилось выше, DML содержит команды манипулирования данными. В этом и следующем разделе рассмотрим только наиболее общие черты команд языка SQL

Фазы выполнения SQL-оператора:

Процесс исполнения SQL-запроса к СУБД можно разделить на 5 основных шагов:

- **Шаг1**

Выполнение синтаксического анализа оператора: проверка соответствия SQL-оператора правилам синтаксиса.

- **Шаг 2**

Проверка параметров запроса: имен таблиц и полей данных, привилегий пользователя к оперируемым объектам, выявление семантических ошибок.

- **Шаг 3**

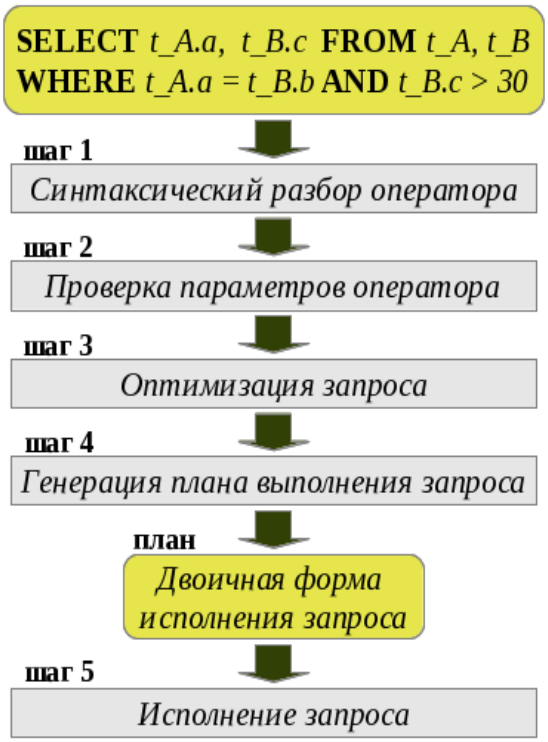
Оптимизация запроса: разбиение запроса на составляющие, оптимизация их выполнения, выбор оптимального плана выполнения с точки зрения состояния БД.

- **Шаг 4**

Генерация двоичного представления оптимизированного запроса (эквивалента двоичного кода программы).

- **Шаг 5**

Выполнение разработанного плана выполнения запроса.



SELECT.

Команда SELECT производит выборку строк из таблиц или представлений. Результатом работы команды является результирующий набор (таблица). При выполнении запроса из приложения, написанного на некотором языке программирования (не SQL) и поддерживающим работу с СУБД, результат выборки сохраняется в памяти приложения или сервера СУБД. Синтаксис команды SELECT, как и весь SQL, формировался постепенно и со временем вобрал в себя огромное количество возможностей. Ниже мы опишем наиболее общие возможности оператора SELECT, описанные в стандарте и поддерживаемые всеми популярными диалектами. В общем виде команда выглядит так:

```
SELECT [ DISTINCT | ALL ] column_list
       FROM table_list
       [[join_type] JOIN join_ conditions]
       WHERE conditions
       [GROUP BY group_list]
       [HAVING group_conditions]
       [ORDER BY order_list [ASK | DESK]]
```

где:

DISTINCT или ALL

используются для указания, должен ли результирующий набор содержать дублирующиеся строки. При указании DISTINCT в результирующем наборе будет присутствовать только одна строка, при указании ALL — все. По-умолчанию подразумевается ALL.

column_list —

список выборки, содержащий имена колонок, разделённых запятыми, которые будут выбраны из указанных таблиц; имена могут выглядеть либо как имя_колонки, либо как имя_таблицы.имя_колонки; если необходимо выбрать все колонки из всех таблиц, можно указать символ *; если в нескольких таблицах содержатся колонки с одинаковыми именами, то в списке выборки такие колонки должны быть квалифицированы именами таблиц; при необходимости, можно переименовать колонку (задать псевдоним) в списке выборки: имя_колонки AS новое_имя.

table_list —

список таблиц или представлений, участвующих в построении результата; здесь должны быть указаны все таблицы и представления, которые используются как в списке выборки, так и в условиях; для имени таблицы также как и для колонки можно задавать псевдоним: имя_таблицы или имя_представления AS новое_имя.

join_type —

указание типа соединения: INNER JOIN (внутренне соединение), LEFT/RIGHT/FULL OUTER JOIN (левое/правое/полное внешнее соединение), CROSS JOIN (перекрестное соединение, декартово произведение).

conditions —

условия, которым должна удовлетворять строка, что бы попасть в результирующий набор записей. Кроме стандартных операций сравнения =, <>/!=, <, >, <=, >= могут использоваться логические операторы OR (или), AND (и) и NOT (логическое отрицание) для соединения нескольких условий; помимо операторов сравнения в условии могут использоваться следующие операторы, детальное описание которых будет дано ниже: BETWEEN (между), IN (в списке), IS NULL (проверка на NULL), EXISTS (существует), LIKE (подобно, проверка на совпадение).

join_conditions —

условия соединения для операторов JOIN. В join_condition могут быть использованы все операторы, которые используются в описанном выше условии conditions.

group_list —

список колонок для GROUP BY, по которым производится группировка записей выборки.

group_conditions —

условия для оператора HAVING, которым должна удовлетворять группа записей, что бы ее строки попали в результирующий набор.

order_list —

список колонок, по которым оператор ORDER BY производит упорядочивание результата выборки, для указания способ сортировки определены два оператора ASC (по возрастанию) и DESC (по убыванию).

Для наглядной демонстрации возможностей языка SELECT разберем его синтаксис на конкретных примерах. Самый простой запрос к таблице или представлению заключается в выборке одного, нескольких или всех столбцов. Для того, чтобы необходимы столбец попал в результирующую выборку необходимо, чтобы он присутствовал в column_list. Результирующая таблица будет содержать столбцы в том порядке, в котором они были указаны в запросе. Соответственно, если вам необходимо иметь в результирующей таблице столбцы в определенной последовательности, то вы должны перечислить столбцы в запросе в нужном вам порядке.

```
SELECT name, address, passport FROM users;
SELECT name, passport, address FROM users;
```

Для того, чтобы не указывать перечисление всех колонок при выборке всех столбцов введен специальный оператор *.

```
SELECT * from users;
```

При работе с несколькими таблицами для исключения конфликта одинаковых имен столбцов, следует указывать имена выбираемых полей с указанием таблицы через точку:

```
SELECT users.name, workers.name from users, workers;
```

В некоторых случаях таблицам, представлениям или колонкам удобно задавать

псевдоним и далее работать уже с ним.

```
SELECT users.name AS user, workers.name AS worker FROM users, workers;
      или
SELECT u.name, w.name FROM users AS u, workers AS w;
```

Самая популярная и востребованная возможность команды SELECT для пользователя — это возможность наложить на выборку некоторое необходимое нам условие, так называемый фильтр. Данная возможность, во-первых, сокращает объем выбираемых данных, так как делает фильтрацию на уровне сервера, во-вторых, сокращает пользователю пост обработку данных.

```
SELECT name, price FROM products WHERE price > 1.6;
```

Более сложные условия могут быть сконструированы используя логические операторы и операторы сравнения.

```
SELECT name, price FROM products WHERE price > 1.6 AND price < 3 AND price <> 4;
```

Для работы с интервалами и массивами данных удобно использовать операторы BETWEEN и IN.

```
SELECT name, price FROM products WHERE price BETWEEN 1.6 AND 3;

SELECT name, price FROM products WHERE price IN (1.6, 2, 2.1, 3);
```

Перед нами часто встают задачи выборки данных по некоторой маски. Например, выбрать все записи для пользователей на букву 'A' или телефонные номера начинающиеся с '+7-383'. Для того рода задач в SQL был введен специальный оператор LIKE. Данный оператор применим только для строковых типов. Каждый диалект имеет свой набор специальных символов (wildcards), которые могут быть использованы при формировании шаблона поиска. Обобщающий символ % поддерживают все диалекты, он представляет последовательность любого числа символов. Так, запрос на поиск всех имен на 'A' будет иметь иметь следующий вид:

```
SELECT name FROM users WHERE name LIKE 'A%';
```

Обратите внимание, что шаблон поиска должен быть заключён в одинарные кавычки.

В реальной жизни заполнение таблиц происходит постепенно, данные которые вносятся зачастую не зависимы друг от друга и не представляют единой отсортированной последовательности. Поэтому таблица представляет неупорядоченный набор данных. Выборка из таких таблиц является также неупорядоченной. Для сортировки в SQL есть специальный оператор ORDER BY с двумя атрибутами ASC и DESC. Сортировать можно по нескольким полям, при этом в начале выборка сортируется по первому столбцу, далее по второму относительно первого и так далее. Давайте рассмотрим, что это означает, на примере сортировки простого массива данных, содержащих имена работников и номеров комнат, где они работают.

```
id        workername       room
 1         Anisyonkov        2
 2         Kim               3
 3         Botov             2
 4         Popov             3
```

Допустим, нам необходимо извлечь всех работников по комнатам, в которых они работают:

```
SELECT room, workername FROM workers ORDER BY room ASC, workername ASC;

Результат:
room    workername
 2        Anisyonkov
 2        Botov
 3        Kim
 3        Popov
```

Если тип сортировки не указан то, по умолчанию используется ASC. Вместо имени столбца, по которому производить сортировку можно указывать его порядковый номер в выборке:

```
SELECT room, workername FROM workers ORDER BY 1, 2;
```

Если есть поля со значением NULL и при этом по этим полям проводиться сортировка, то тогда положение таких строк в выборке может быть произвольным. Остальные строки будут отсортированы (без учета строк с NULL). Часто такие строки помещаются в начало или в хвост выборки.

Рассмотрим на примере таблиц A и B, представленных ниже, несколько примеров использования команды SELECT.

BBA

A	
NUM	**STR_A**
1	aaa
2	asd
3	qwe
7	zaq

B	
NUM	**STR_B**
2	aaa
2	aaa
4	main
5	blue

1. Получить все пары чисел, которым соответствуют совпадающие строки. (Под строкой в данных примерах подразумеваются значения в колонках A.STR_A и B.STR_B):

```
SELECT A.NUM, B.NUM FROM A, B WHERE STR_A=STR_B;
```

2. Получить записи из A, с номером больше 2:

```
SELECT * FROM A WHERE NUM > 2;
```

3. Получить прямое произведение двух таблиц, упорядоченное по номерам из B, а в случае из равенства, по номерам из A:

```
SELECT * FROM A, B ORDER BY B.NUM, A.NUM;
```

4. Получить из B сумму чисел для записей, имеющих одинаковые строки, при условии, что сумма меньше 5:

```
SELECT * FROM B GROUP BY STRB HAVING SUM(NUM) > 5;
```

При условии, что две команды SELECT возвращают результирующие наборы с заголовками, совпадающими по типа данных, над результирующими наборами можно произвести операции объединения, пересечения и разности. Например:

```
SELECT NUM, STR FROM A       UNION  SELECT NUM, STRB AS STR FROM B
```

Внешнее объединение.

Многие современные реляционные СУБД вводят понятие *внешнего объединения*, которое отсутствует в реляционной модели. Рассмотрим для примера такие таблицы, являющиеся модификацией отношений, представленных на странице 9:

A	
NUM	**STR_A**
1	aaa
2	asd
3	qwe
7	zaq

B	
NUM	**STR_B**
2	aaa
2	aab

В случае, если мы выполним запрос

```
SELECT * FROM A, B WHERE A.NUM=B.NUM
```

мы получим результат вида:

A.NUM	A.STR_A	B.NUM	B.STR_B
2	asd	2	aaa

2	asd	2	aab

Т.е. мы получили классическое экви-соединение. Однако, на практике часто возникает необходимость получить **все** строки из A, даже если им нет соответствующих строк в B. Такое соединение принято называть *внешним*. Используя синтаксис СУБД Oracle его можно записать таким образом:

```
SELECT * FROM A, B WHERE A.NUM=B.NUM(+)
```

Результатом данной команды будет следующий набор:

A.NUM	A.STR_A	B.NUM	B.STR_B
1	aaa		
2	asd	2	aaa
2	asd	2	aab
3	qwe		
7	zaq		

INSERT.

Команда INSERT используется для вставки данных в таблицу базы данных и имеет две возможных записи.

```
INSERT INTO table_name [(column_list)] VALUES (value_list)
INSERT INTO table_name [(column_list)] select_statement
```

где:

table_name —

имя таблицы, в которую производится вставка;

column_list —

список колонок, в которые производится вставка;

value_list —

список значений для вставки, порядок значений должен соответствовать порядку колонок в column_list, если список колонок опущен, то подразумевается, что value_list содержит данные для всех полей, и данные следуют в том же порядке, в каком они были указаны при создании таблицы;

select_statement —

указывается предложение SELECT, которое возвращает набор записей, каждую из которых можно трактовать как value_list.

UPDATE.

Команда UPDATE используется для модификации данных в таблице.

```
UPDATE table_name SET (column_list) = (select_statement_1) WHERE condition
```

или

```
UPDATE table_name SET column_name = expr | (select_statement_2) [, ... ] WHERE condition
```

здесь:

table_name —

имя таблицы, в которую производится вставка;

column_list —

список колонок, в которых производятся изменения;

select_statement_1 —

указывается предложение SELECT, которое возвращает набор записей, содержащих новые значения для колонок;

column_name —

имя колонки, в которую будут вносится изменения;

expr —

выражение, или

select_statement_2 —

предложение SELECT, которые содержат значение для указанной колонки;

| (вертикальная черта между expr и select_statement_2)

символизирует, что должно быть указанно либо выражение, либо предложение SELECT;

, … (запятая и многоточие в квадратных скобках)

означают, что на их месте может быть указано произвольное количество элементов, вида column_name = expr | (select_statement_2), разделённых между собой запятыми.

DELETE.

Команда DELETE используется для удаления данных из таблицы.

```
DELETE FROM table_name WHERE condition
```

Прочие DML команды.

В зависимости от конкретного диалекта, SQL может поддерживать дополнительные команды. Например, команда REPLACE в MySQL:

```
REPLACE INTO table_name VALUES ( value_list );
```

value_list обязательно должен содержать первичный ключ. Данная команда проводит в таблице поиск по первичному ключу, если данные были найдены, то для них выполняется операция модификации, аналогично UPDATE. Если же данных с указанным первичным ключом не существует, то выполняется операция вставки.

Команда MERGE в Oracle обладает существенно более широким функционалом, но, в целом, служит для тех же целей.

Команды PURGE в Oracle и TRUNCATE в PostgreSQL служат для очень быстрого удаления всех данных в таблице.

DDL.

Подъязык DDL содержит команды для создания, модификации и удаления объектов словаря базы данных, а также команды для предоставления системных и объектных привилегий, определяющих права доступа к данным объектам.

В рамках данного раздела, в их наиболее общем виде, мы рассмотрим только команды для работы с таблицами и представлениями.

CREATE TABLE

```
CREATE TABLE table_name
( col_name1    col_type1      [DEFAULT default_expr1]      [constraint1]
, col_name2    col_type2      [DEFAULT default_expr2]      [constraint2]
,...
, table_constraint1
, table_constraint2
,...
);
```

либо

```
CREATE TABLE table_name AS select_statement;
```

где

table_name

имя создаваемой таблицы;

col_name

имя атрибута;

col_type

тип данных данного атрибута;

default_expr

указывает значение по-умолчанию для данного атрибута;

constraint

ограничение целостности данного атрибута;

table_constraint

ограничение целостности, налагаемое на строку таблицы в целом (например, сумма двух атрибутов не должна превышать некоторого значения);

select_statement

атрибуты новой таблицы будут в точности соответствовать списку выборки предложения SELECT, кроме того, таблица будет содержать все записи, которые вернет данное предложение;

*****

таблица может содержать любое количество атрибутов[10];

*****

таблица может содержать любое количество элементов table_constraint.

ALTER TABLE и DROP TABLE.

```
ALTER TABLE table_name
       ADD
       ( col_name1    col_type1      [DEFAULT default_expr1]       [constraint1]
       , col_name2    col_type2      [DEFAULT default_expr2]       [constraint2]
       ,...
       , table_constraint1
       , table_constraint2
       ,...
       )
       MODIFY
       ( col_name3 [col_type3]       [DEFAULT default_expr3]       [constraint3]
       ,...
       )
       RENAME COLUMN old_name TO new_name
       DROP ( col_name4, col_name5, ... )
;
```

где фраза *ADD* содержит описание добавляемых к таблице полей и ограничений целостности, аналогично тому, как это описано в разделе CREATE TABLE, а фраза *MODIFY* — модифицирует уже существующие поля.

Иногда СУБД не позволяют удалять колонки из таблиц, т.к. при этом возникают существенные накладные расходы на перестройку физической структуры таблицы. Проблему удаления колонки решают двумя способами:

1. создается представление, у которого в списке выборки указаны все поля, кроме нежелательного; после чего, вся работа выполняется через представление[11];
2. создается временная таблица, в которую копируются все данные, основная таблица удаляется и создается вновь, но уже без нежелательного поля, во вновь созданную таблицу переносятся данные из временной таблицы.

Минусом первого подхода является определенный расход дискового пространства, для хранения NULL-значений в нежелательном поле, и необходимость использовать новое имя (имя представления) при работе с данными, что может привести к необходимости модифицировать прикладные программы[12].

Основным минусом второго подхода являются высокие накладные расходы при

10 На самом деле, каждая конкретная СУБД накладывает определенные ограничения на количество атрибутов в таблице, но как правило, эти число достаточно большое. Например, таблицы в Oracle 7.3 не могут содержать более 254 атрибутов.

11 При этом желательно все значения данного поля установить в NULL, а также установить ограничение целостности, которое запрещало бы вставку значений отличных от NULL

12 К сожалению, переименовать таблицу также не всегда возможно

создании временной таблицы и копировании данных из временной таблицы во вновь созданную основную таблицу, а также необходимость заново определять все права доступа к созданной таблице.

Удаление таблицы выполняется командой

```
DROP TABLE table_name;
```

В случае, если существуют таблицы, ссылающиеся на первичный ключ удаляемой таблицы, то, как правило, СУБД не позволит удалить указанную таблицу, даже если удаляемая таблица пуста.

CREATE и DROP VIEW.

В соответствии со стандартом SQL92 для представлений определяются две команды: CREATE VIEW для создания представления и DROP VIEW для его удаления.

```
CREATE VIEW view_name AS select_statement;
DROP VIEW view_name;
```

Хотелось бы обратить внимание на тот факт, что при необходимости переопределить представление, представление необходимо удалить и создать заново. При этом все права доступа на представление теряются, и должны быть выданы заново. Во избежание этой процедуры некоторые СУБД (в частности Oracle) позволяют указывать ключевую фразу CREATE OR REPLACE VIEW, которая либо создает новое представление, либо переопределяет уже существующее.

Еще один интересный момент, связанный с представлениями, возникает, когда удаляется таблица, на которой основан базовый SELECT.

Возможны два варианта поведения СУБД в такой ситуации:

1. отказаться удалять таблицу;
2. удалить таблицу, и пометить представление как неправильное *invalid*, а при обращениях к неправильному представлению пользователю выдавать соответствующую ошибку.

СУБД Oracle реализует второй вариант с некоторыми дополнениями: если при очередном обращении к представлению возникает ошибка доступа к данным – это может быть отсутствие таблицы, изменение её структуры таким образом, что базовый SELECT становится синтаксически неверным, потеря прав доступа к таблице другого пользователя,..– представление помечается как *invalid.* Представление остаётся неправильным даже после того, как проблема с доступом будет решена (например, будет восстановлена таблица, используемая в базовом SELECT'е), и требует либо пересоздания, либо выполнения команды

```
ALTER VIEW view_name RECOMPILE;
```

Процедурные языки баз данных.

Язык SQL, являясь непроцедурным языком программирования, очевидно, не является самодостаточным средством при создании приложений, работающих с СУБД. Как правило, это обходится путём встраивания SQL в программы, написанные на каком-нибудь стандартном процедурном языке: C/C++, Java, Perl, Ada и т. п.

Такой подход снимает многие ограничения SQL, вызванные его непроцедурностью, но существуют и чисто технические проблемы при использовании связки "процедурный язык" + SQL. В качестве типичного примера можно привести экспертную систему, работающую по модели "клиент-сервер". При такой схеме, клиент запрашивает данные из СУБД, обрабатывает их в соответствии с бизнес-логикой приложения и выдаёт результаты работы конечному пользователю. При построении большой экспертной системы объёмы передаваемых по сети данных могут быть весьма значительны, что является серьёзным барьером при наращивании производительности системы.

Учитывая выше сказанное, многие производители СУБД разрабатывают специальные механизмы, позволяющие перенести бизнес-логику приложения на сторону СУБД. Иными словами, создавать и хранить процедуры и функции в рамках базы данных. Такие процедуры и функции принято называть хранимые. Процедурные языки программирования, используемые для создания этих программных компонент могут быть как специально созданными: PL/SQL у Oracle, pgPLSQL у PostgreSQL, так и вполне распространёнными языками программирования: Java, Perl, Python и т. п. Основное преимущество специально разработанных языков программирования состоит в том, что они включают SQL как свое подмножество.

В приведённом выше примере взаимодействие клиента и сервера может свестись к тому, что клиент произведёт вызов хранимой процедуры, сервер, выполняя данную процедуру, обработает данные (обмен данными при этом будет происходить в оперативной памяти, что, очевидно, быстрее чем передача данных по сети) и выдаст результаты (например, заполнит результирующую таблицу), клиент получит результаты от сервера и выдаст их пользователю.

В качестве примера в данной главе будет рассматриваться язык PL/SQL, созданный корпорацией Oracle, и используемый в ее продуктах. В качестве прототипа языка PL/SQL был выбран языка Ада.

Обзор PL/SQL.

Базовая единица языка PL/SQL представляет из себя набор блоков. (В частном случае — один блок.) Блоки могут быть как именованными, так и анонимными (неименованными). Общий вид именованного блока выглядит так:

```
<<block_name>>
DECLARE
       /*раздел объявления переменных*/
BEGIN
       /*тело блока*/
EXEPTION
       /*раздел обработчиков исключений*/
END block_name;
```

При этом обязателен только раздел BEGIN, разделы DECLARE и EXEPTION могут быть опущены. При создании анонимного блока метка <<block_name>> и block_name в предложении END опускаются[13].

Идентификаторы.

Идентификатор в языке PL/SQL начинается с буквы и может содержать до 30 букв

13 Синтаксис создания процедур и функций будет рассмотрен в последующих разделах

латинского алфавита, цифр, знаков подчеркивания (_), знаков доллара ($) и знаков диез (#). Регистр букв в PL/SQL, как и в диалекте SQL, используемом продуктами фирмы Oracle, не различается. Однако, если существует необходимость сделать идентификатор чувствительным к регистру, или поместить в него недопустимый символ, то идентификатор необходимо заключить в двойные кавычки.

Литералы.

Литерал — это символьное, числовое или логическое значение, не являющееся идентификатором.

Символьные.

Символьные (или строковые) литералы состоят из одного или нескольких символов, ограниченных одинарными кавычками. В случае если литерал должен содержать одинарную кавычку, на ее месте пишется две одинарных кавычки подряд. Считается, что символьные литералы имеют тип CHAR.

Числовые.

Числовые литералы состоят из цифр, перед которыми могут стоять знаки + и -, и могут включать десятичную точку. Можно также указывать экспоненциальное представление aE[+|-]b = $a*10^{\pm b}$. При этом значение экспоненты должно быть обязательно целым.

Логические.

В PL/SQL существует три логических литерала: TRUE, FAULSE и NULL.

Объявление и инициализация переменных. Типы данных.

Общий вид объявления переменной выглядит следующим образом:

```
var_name data_type [CONSTANT] [NOT NULL] [DEFAULT value]
```

где

var_name — имя создаваемой переменной;

data_type — ее тип данных;

CONSTANT — будет ли данная переменная константой[14];

NOT NULL — данный параметр указывает, что переменная не может содержать NULL-значения;

DEFAULT — указывает инициализирующее значение для переменной, слово DEFAULT можно заменить на знак :=

Все неинициализированные переменные имеют значение NULL.

При использовании PL/SQL необходимо помнить, что типы данных языка PL/SQL неточно отображаются на типы данных языка SQL. Так например, тип данных VARCHAR2 в SQL имеет максимальную длину 2000 байт для Oracle7 и 4000 байт для Oracle8, но в PL/SQL переменная этого типа позволяет хранить до 32 767 байт.

BINARY_INTEGER.

Данные с типом данных NUMBER в СУБД Oracle хранятся в специальном десятичном формате, который был разработан для точного и эффективного хранения информации. По этому при выполнении арифметических операций производится преобразование значений типа NUMBER к двоичному виду.

Для различного рода счётчиков такие преобразования приводят к значительным накладным расходам. Специально для этого в PL/SQL существует тип BINARY_INTEGER, используемый для хранения знаковых целых чисел в диапазоне ±2 147 483 647.

BOOLEAN.

14 как и при описании синтаксиса SQL в предыдущей главе, условимся, что необязательные параметры команды указываются в квадратных скобках

В отличие от SQL, PL/SQL содержит логический тип BOOLEAN, который допускает три значения переменной: TRUE, FALSE и NULL.

Ссылки.

После создания переменной, PL/SQL допускает создание еще одной переменной, которая содержит адрес первой. Тем самым, ссылки являются точным аналогом указателей в классических языках программирования.

При создании большого проекта возникает проблема согласования типов данных между атрибутами таблицы и переменными программы. Самый простой способ решения такой проблемы — использование атрибута %TYPE. Например,

```
var_name table_name.column_name%TYPE;
```

создаст переменную с тем же типом данных, что и атрибут col_name в таблице table_name.

Команды PL/SQL.

PL/SQL содержит вполне типичный набор команд, мало отличающийся от таких процедурных языков, как C или Pascal.

Рассмотрим основные операторы языка.

Условный оператор.

```
IF bool_expr1 THEN
       operator_sequence1;
ELSIF bool_expr2 THEN
       operator_sequence2;
ELSE
       operator_sequence3;
END IF;
```

Оператор позволяет указать произвольное количество конструкций ELSIF...THEN. Обязательной конструкцией является только IF...THEN, конструкции ELSIF и ELSE — необязательны.

Оператор ветвления.

Существует два варианта записи этого оператора. Можно указать исчисляемое выражение после фразы CASE, а после каждой фразы WHEN указывать только константное значение:

```
CASE condition_expr
       WHEN value_1 THEN
              stmt_1;
       ...
       WHEN value_n THEN
              stmt_n;
       ELSE
              default_stmt;
END CASE;
```

Либо же, можно присвоить значение выражения какой-то переменной, после CASE ничего не указывать, а после каждого WHEN проверять данную переменную на равенство какой-либо константе:

```
var_name := condition_expr;
CASE
       WHEN var_name = value_1 THEN
              stmt_1;
       ...
       WHEN var_name = value_n THEN
              stmt_n;
       ELSE
              default_stmt;
END CASE;
```

Если при в операторе CASE фраза ELSE была опущена, а на момент исполнения программы ни одна из секций WHEN не выполнилась, Oracle генерирует исключение

CASE_NOT_FOUND, которое должно быть обработано:

```
<<example>>
DECLARE
        . . .
BEGIN
        . . .
        CASE condition_expr
                WHEN value_1 THEN
                        stmt_1;
                ...
                WHEN value_n THEN
                        stmt_n;
        END CASE;
        . . .
EXCEPTION
        WHEN CASE_NOT_FOUND THEN
                . . .
END example;
```

Циклы.

Простейший цикл в PL/SQL образуется следующим образом:

```
LOOP
        operator_sequence;
END LOOP;
```

Вообще говоря, это бесконечный цикл, но последовательность операторов может содержать оператор EXIT, при достижении которого, выполнение цикла прервётся. Если оператор EXIT должен быть исполнен только при выполнении некоторого условия, то данное условии можно указать так:

```
EXIT WHEN bool_expr;.
```

При необходимости прервать исполнение текущей итерации цикла и сразу перейти к следующей, используется оператор CONTINUE (CONTINUE WHEN ...).

Еще один вариант цикла — цикл WHILE

```
WHILE bool_expr LOOP
        operator_sequence;
END LOOP;
```

будет выполняться до тех пор, пока выражение bool_expr остается истинным.

```
FOR var_counter IN [REVERSE] min_val..max_val LOOP
        operator_sequence;
END LOOP;
```

Это классический оператор for. Переменная var_counter может не описываться в разделе DECLARE, но тогда область видимости переменной будет составлять тело цикла. Если необходимо, чтобы цикл выполнялся от большего значения к меньшему, указывается ключевое слово REVERSE, при этом границы диапазона всегда указываются в порядке минимальное_значение .. максимальное_значение.

Для повышения читаемости кода программы циклы можно именовать. Например:

```
<<loop_name>>
FOR v IN min_v .. max_v LOOP
        ...
END LOOP loop_name;
```

Именование циклов и объемлющих блоков позволяет обратиться к их переменным, даже если их имя совпадает с именем счётчика в цикле:

```
<<main>>
DECLARE
        i NUMBER : = 1;
BEGIN
        <<outer_loop>>
        FOR i IN 1 .. 100 LOOP
                <<inner_loop>>
                FOR i IN 1 .. 30 LOOP
                        DBMS_OUTPUT.PUT_LINE( 'inner: ' || TO_CHAR(i) );
                        DBMS_OUTPUT.PUT_LINE( 'outer: ' || TO_CHAR(outer.i) );
                        DBMS_OUTPUT.PUT_LINE( ' main: ' || TO_CHAR(main.i) );
                END LOOP inner_loop;
        END LOOP outer_loop;
```

```
END main;
```

GOTO

Он есть!

```
LOOP
       . . .
       IF bool_expr THEN
              GOTO goto_lable;
       END IF;
END LOOP;

. . .

<<goto_lable>>
some_stmt;
```

Необходимо учитывать, что метка может указываться только перед блоком или предложением, но не внутри предложения. Т. е. данный код ошибочен:

```
. . .
<<lable>>
END LOOP;
. . .
```

NULL-оператор

Везде, где по синтаксису языка должно стоять предложение, но указать такое предложение мы не можем, используется оператор NULL. Вот наиболее типичные области применения этого оператора:

```
. . .
IF is_ok THEN
       -- тут мы будем делать всем «счастье», но пока не знаем как
       NULL;
END IF;


LOOP
       . . .
       <<label_before_end>>
       NULL;
END LOOP
```

Составные типы данных.

Аналогично другим процедурным языкам программирования PL/SQL позволяет создавать составные типы данных, определяемые пользователем. В их число входят: записи, таблицы PL/SQL и объекты. Рассмотрение последних выходит за рамки данной методички.

Записи.

Создание записи происходит в разделе описания переменных, и выглядит следующим образом:

```
TYPE record_type_name IS RECORD
       ( field1 type1 [NOT NULL] [DEFAULT expr1]
       , field2 type2 [NOT NULL] [DEFAULT expr2]
       , ...
       );
rec_var record_type_name;
```

Описание полей записи аналогично описанию простых переменных. Присваивание записей между собой допустимо только тогда, когда их типы совпадают, нельзя присвоить запись одного типа записи другого типа, даже если описания этих типов совпадают. Для обращения к конкретному полю записи необходимо написать так: rec_var.field_name

Записи часто используются для обработки данных, возвращаемых оператором SELECT. Для этого синтаксис оператора SELECT, по сравнению с рассмотренным в предыдущей

главе, расширяется конструкцией INTO var_list, где под var_list подразумевается список переменных, в которые записываются данные, возвращаемые SELECT'ом. Такой вариант команды SELECT должен возвращать не более одной записи, т. к. в противном случае возникает соответствующая ошибка периода исполнения.

Рассмотрим пример:

```
<<example>>
DECLARE
       employer employers%ROWTYPE;
BEGIN
       ...
       SELECT * FROM employers INTO employer WHERE ID = 10000;
       ...
END example;
```

Данный пример иллюстрирует использование атрибута %ROWTYPE, который применим к таблицам, записям и курсорам. Его назначение аналогично назначению атрибута %TYPE.

Ассоциативные массивы (таблицы PL/SQL).

Общий синтаксис описания таблицы выглядит так:

```
TYPE table_type_name IS TABLE OF base_type INDEX BY BINARY_INTEGER;
tab_var table_type_name;
```

Индексация таблицы PL/SQL может выполняться либо целочисленными, либо строковыми типами.

Обращение к i-му элементу таблицы происходит так: tab_var(i), где i — это выражение имеющее тип данных, как указан во фразе INDEX BY, либо приводимое к этому типу. Для указанного выше примера это должен быть либо BINARY_INTEGER, либо другой целочисленный тип данных.

Присваивание i-му элементу значения создаёт этот элемент (если его еще не было), а при обращении к индексу, которого ещё нет в таблице, Oracle генерирует ошибку ORA-1403: no data found. Память под таблицу отводится динамически — по мере необходимости.

Если таблица содержит записи, то обращение к полю i-ой записи выглядит так: tab_name(i).field_name.

Все таблицы в PL/SQL обладают набором атрибутов, описанных в следующей таблице:

Имя атрибута	Возвращаемый тип	Описание
COUNT	NUMBER	Количество элементов, содержащихся в таблице
DELETE	—	Удаляет элементы из таблицы
EXISTS	BOOLEAN	Возвращает TRUE, если элемент существует
FIRST	BINARY_INTEGER	Возвращает индекс первого элемента
LAST	BINARY_INTEGER	Возвращает индекс последнего элемента
NEXT	BINARY_INTEGER	Возвращает индекс элемента, следующего за указанным
PRIOR	BINARY_INTEGER	Возвращает индекс элемента, предшествующего указанному

Данный пример иллюстрирует работу с атрибутами таблицы.

```
<<example>>
DECLARE
       index BINARY_INTEGER;
       TYPE tt IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;

       table_var tt;
BEGIN /*инициализируем таблицу*/
       <<tab_init>>
       FOR index IN 1 .. 10 LOOP
              table_var(index) := index;
```

```
        END LOOP tab_init;
        /*встаём на первый элемент таблицы*/
        index := table_var.FIRST;
        /*встаем на второй элемент таблицы*/
        index := table_var.NEXT(index);
        /*удаляем второй элемент таблицы*/
        IF table_var.EXISTS(index) THEN
               table_var.DELETE(index);
        END IF;
        /*удаляем с 4-го по 7-й элемент*/
        table_var.DELETE(4,7);
        /*удаляем все содержимое таблицы*/
        table_var.DELETE;
END example;
```

Хранимые (вложенные) таблицы

Это набор однотипных значений, который можно хранить в таблице БД. Концептуально, это массив с произвольным числом элементов.

```
TYPE nest_table IS TABLE OF VARCHAR2(30);
```

Массив

```
TYPE Calendar IS VARRAY (366) OF DATE;
```

Может хранить, в данном случае, до 366 элементов. VARRAY массивы являются точными аналогами массивов из классических языков программирования.

Курсоры.

При выполнении SQL-оператора Oracle создаёт область памяти, называемую *контекстной областью*. Она содержит информацию, необходимую для завершения обработки запроса, например:

- число строк, обрабатываемых запросом;
- указатель на представление оператора после синтаксического разбора;
- *активный набор* — множество строк, возвращаемых запросом.

***Курсор* — это указатель на контекстную область.**

Курсоры бывают явные и неявные. Последние возникают при выполнении, например, операторов SELECT...INTO, или UPDATE. Для работы с явным курсором необходимо выполнить четыре операции:

1. объявить курсор
2. открыть курсор для запроса
3. провести выборку результатов
4. закрыть курсор

Объявление и открытие курсора.

```
DECLARE
        CURSOR cr IS select_statement;
BEGIN
        OPEN cr;
 ...
```

Предложение SELECT в объявлении курсора не может содержать конструкцию INTO, т. к. выборка данных производится оператором FETCH. При открытии курсора Oracle'ом выполняется:

- анализ значений переменных привязки;
- на их основе выделяется активный набор;
- указатель активного набора ставится на первую строку.

Выборка результатов и закрытие курсора.

На приведенном примере видно как происходит выборка данных из открытого курсора.

```
BEGIN
       OPEN cr;
       <<fetch_loop>>
       WHILE cr%FOUND LOOP
              FETCH cr INTO var_list;
              ...
       END LOOP fetch_loop;
       CLOSE cr;
```

var_list должен содержать либо список переменных, в которые записываются значения, возвращаемые SELECT'ом, либо указывается переменная-запись PL/SQL. Оператор FETCH выбирает одну строку за одно обращение. При таком чтении результатов часто говорят, что курсор "сдвигается" на следующую строку в активном наборе[15]. Однако, нет никакой возможности "вернуть" курсор на предыдущую строку активного набора.

Закрытие курсора выполняется оператором CLOSE.

При закрытии курсора все ресурсы, с ним связанные, освобождаются.

С каждым курсором связаны четыре атрибута:

%FOUND

проверяет, была ли выбрана строка с данными, если да, то возвращается TRUE;

%NOTFOUND

обратен к %FOUND;

%ISOPEN

проверяет, был ли открыт курсор;

%ROWCOUNT

возвращает количество строк, уже прочитанных через курсор.

Атрибуты %FOUND, %NOTFOUND и %ROWCOUNT применимы только к открытому курсору.

При обработке INSERT, UPDATE, DELETE и SELECT INTO PL/SQL сам создаёт и обрабатывает курсор, по-этому операции OPEN, FETCH и CLOSE не нужны. Однако, к таким неявным курсорам применимы курсорные атрибуты. Например:

```
UPDATE ... SET ... WHERE ...;
IF SQL%NOTFOUND THEN
... /*делаем что-то, если нет данных для UPDATE*/
END IF;
```

Использование %NOTFOUND применительно к предложениям SELECT INTO синтаксически корректно, но бессмысленно, т. к. если строки не были выбраны, то порождается ошибка ORA-1403: no data found, и управление передаётся обработчику ошибок, если таковой есть. Соответствующее исключение называется NO_DATA_FOUND.

Курсорный FOR.

В PL/SQL существует возможность организовать цикл FOR, проходящий по всем значениям, возвращаемым курсором.

```
<<example>>
DECLARE
       CURSOR c IS SELECT ...;
BEGIN
       FOR r IN c LOOP
              /*обработка результатов SELECT'a*/
       END LOOP;
END example;
```

При этом необходимо обратить внимание на два момента: а) переменная r не описывается в разделе DECLARE, ее область действия — тело оператора FOR, и б) операторы OPEN, FETCH и CLOSE выполняются для c неявно.

15 Вообще, на практике принято считать, что курсор указывает не на контекстную область, а на тут или иную строку активного набора. Что, строго говоря, не правильно, но наглядно демонстрирует назначение курсора.

Параметризованные курсоры.

При создании курсора в конструкциях WHERE и HAVING можно указывать так называемые *переменные привязки* (binding variables). Их использование продемонстрировано на примере.

```
<<example>>
DECLARE
       var NUMBER;
       CURSOR c IS SELECT ... FROM tab WHERE tab.col = var ;
BEGIN
       var := 1;
       OPEN c;
       ...
END example;
```

Основным недостатком такой техники является неочевидность связи между переменной var и курсором c. Еще один недостаток — заведение переменных в которых, вообще говоря, нет необходимости.

Для избавления от этих недостатков используются *параметризованные курсоры*. Пример такого курсора приводится ниже.

```
<<example>>
DECLARE
       CURSOR c(var NUMBER) IS SELECT ... FROM tab WHERE tab.col = var ;
BEGIN
       OPEN c(1);
       ...
END example;
```

Т. е., тем самым мы избавляемся от лишних переменных, а значения для переменных привязки при открытии курсора указываются явно, что уменьшает вероятность ошибок. В остальном же параметризованные курсоры ведут себя в точности также, как и обычные непараметризованные.

Курсорные переменные.

Курсорные переменные используются для передачи курсора как параметра в подпрограммы, т. к. курсоры передавать как параметр в подпрограммы нельзя. Их можно представлять как «указатели» на результирующий набор.

Курсорные переменные бывают ограниченные (strong или constrained) и неограниченные (week или unconstrained). В первом случае тип возвращаемого значения определяется фразой RETURN. Во втором случае таких ограничений нет. (Продолжая аналогию с указателями: типизированные и нетипизированные указатели).

```
<<example>>
DECLARE
   -- Strong/constrained
   TYPE s_cur_t IS REF CURSOR RETURN t_name%ROWTYPE;

   -- Weak/unconstrained
   TYPE w_cur_t IS REF CURSOR;

   cursor1 s_cur_t;
   cursor2 w_cur_t;
   my_cursor SYS_REFCURSOR; -- no new type needed

BEGIN
. . ......
END example;
```

SELECT FOR UPDATE.

В операторе SELECT можно указать конструкцию FOR UPDATE [OF column_list], которая устанавливает блокировки на изменение для строк активного набора. При создании курсора на основе такого предложения появляется возможность изменять данные, находящиеся в строке под курсором. Такие изменения производятся командой WHERE CURRENT OF cursor_name].

Единственной тонкостью использования такой техники является не возможность использования команды COMMIT в цикле выборки-модификации, т. к. COMMIT снимает все блокировки сеанса, в том числе и блокировки, установленные командой SELECT FOR UPDATE. При необходимости использования команды COMMIT в цикле, единственный способ решения проблемы — использование первичного ключа.

Процедуры и функции PL/SQL.

Создание процедуры или функции в языке PL/SQL наглядно демонстрируется следующим примером:

```
CREATE [OR REPLACE] PROCEDURE proc_name
       [(arg1 [IN|OUT|IN OUT] arg1_type
       , arg2 [IN|OUT|IN OUT] arg2_type
       , ... ) ]
IS | AS
       /*раздел объявления переменных */
       /*и вложенных процедур и функций*/
BEGIN
       /*тело процедуры*/
END proc_name;

CREATE [OR REPLACE] FUNCTION func_name
       [(arg1 [IN|OUT|IN OUT] arg1_type
       , arg2 [IN|OUT|IN OUT] arg2_type
       , ... ) ]
RETURN ret_type
IS | AS
       /*раздел объявления переменных */
       /*и вложенных процедур и функций*/
BEGIN
       /*тело функции*/
END func_name;
```

Как обычно, квадратные скобки указывают на необязательность их содержимого, многоточие — допустимость указания произвольного количества аргументов, а вертикальная черта (|) — обязательное указание одного из элементов, перечисленных через неё.

Необходимо учитывать, что при описании параметров нельзя накладывать ограничения на длину типов CHAR и VARCHAR2, на точность и масштаб типа NUMBER, т. к. ограничения на формальные параметры процедуры или функции накладываются на момент вызова фактическими параметрами. В случае крайней необходимости это требование языка можно обойти, используя атрибут %TYPE применительно к переменной PL/SQL или колонке таблицы базы данных, имеющей нужный тип данных.

Описывая формальный параметр процедуры или функции можно указать *вид* параметра, т. е. указать, будет ли данный параметр внутри тела процедуры доступен только на чтение (IN), только на запись (OUT), или на чтение-запись (IN OUT). По-умолчанию, считается, что параметры имеют вид только на чтение.

При вызове процедуры или функции, формальные и фактические параметры связываются либо позиционно (в точности, как в классических языках C или Pascal), либо поимённо. Допустим, что существует процедура Proc(a1 NUMBER, a2 NUMBER, a3 NUMBER), и переменные v1, v2, v3 типа NUMBER. Согласно вышесказанному, вызов процедуры можно записать двумя способами:

```
Proc(v1, v2, v3);
```

либо

```
Proc(a1 => v1, a2 => v2, a3 => v3);
Proc(a2 => v2, a3 => v3, a1 => v1);
```

Модули.

С целью логического объединения нескольких процедур или функций, PL/SQL позволяет создавать программные модули[16]. В PL/SQL модуль состоит из двух частей —

16 Альтернативное название — пакеты.

заголовка модуля (package header, package specification) и тела модуля (package body).

Заголовок содержит информацию о составе модуля, но не содержит реализацию подпрограмм. Напротив, тело модуля содержит реализацию подпрограмм, объявленных в заголовке. Любое объявление подпрограммы должно быть раскрыто в теле модуля.

Внутри модуля допускается переопределение процедур и функций. Т. е. допускается существование нескольких подпрограмм с одинаковым именем, но различными параметрами. При этом недопустимо, чтобы списки формальных параметров отличались только именами или видами параметров. Также недопустимо, чтобы функции отличались только типами возвращаемых значений. Последнее требование — типы формальных параметров должны принадлежать к различным семействам[17], т. е. недопустимо переопределение процедуры у которой один и тот же аргумент в одном случае принадлежит типу CHAR, а в другом — VARCHAR2[18].

Триггеры.

Многие СУБД, в том или ином виде, поддерживают понятие *триггера*, который представляет из себя хранимую процедуру без параметров, вызываемую СУБД автоматически в случае того или иного события (например, выполнение команды UPDATE над таблицей).

На основе триггеров обычно реализуются сложные ограничения целостности, которые невозможно выразить средствами SQL, протоколирование операций над таблицей и т. п.

Общий синтаксис создания триггера выглядит так:

```
CREATE [OR REPLACE] TRIGGER trg_name
       BEFORE | AFTER event [OR event ...]
       ON table_name [FOR EACH ROW [WHEN expr]]
BEGIN
       /*тело триггера*/
END trg_name;
```

Под event понимается событие, на которое должен сработать триггер. Допустимыми значениями являются: INSERT, UPDATE[19], DELETE и INSTEAD OF. При создании триггера можно указать несколько условий, перечислив из через ключевое слово OR.

Триггеры разделяются на строковые (row trigger) и операторный (statement trigger). Первые из них создаются указанием конструкции FOR EACH ROW, и выполняются для каждой строки, которую обрабатывает предложение SQL. Вторые — выполняются только один раз на каждую инициировавшую их команду SQL.

Для строковых триггеров можно указать условие их срабатывания (expr в приведенном выше примере создания триггера). При его указании, перед вызовом триггера будет вычисляться указанное условие, и только в случае его истинности будет выполняться тело триггера.

Так же для строковых триггеров определены две псевдозаписи :old и :new, имеющие тип table_name%ROWTYPE. Данные псевдозаписи содержат старое значение строки и новое значение, которое вносит предложение SQL в таблицу. Изменяя значения псевдозаписи :new, можно внести в таблицу данные, отличающиеся от указанных предложением SQL. Так, например, можно построить присвоение уникального номера строке таблицы.

При выполнении команды SQL над таблицей триггеры всегда срабатывают в следующем порядке:

1. Операторный триггер BEFORE
2. Строковый триггер BEFORE
3. Оператор SQL

17 Разделение типов по семействам подробно описано в „Oracle SQL Refence”.

18 На самом деле компилятор позволит создать подпрограммы, нарушающие эти ограничения, но при обращении к ним случится ошибка.

19 В случае UPDATE, можно указать UPDATE OF и список колонок, при модификации которых будет срабатывать триггер

4. Строковый триггер AFTER
5. Операторный триггер AFTER

Если для таблицы определено несколько триггеров одинакового типа, например, два триггера BEFORE UPDATE, порядок их срабатывания не определяется.

Внутри тела триггера определены три предиката, принимающие истинное значение в зависимости от оператора SQL, активировавшего данный триггер. Рассмотрим, например, простейший триггер, регистрирующий изменения данных в таблице.

```
CREATE OR REPLACE TRIGGER logger
       AFTER INSERT OR UPDATE OR DELETE
       ON base_table
DECLARE
       operation CHAR(1);
BEGIN
       IF INSERTING THEN
              operation := 'i';
       ELSIF UPDATING THEN
              operation := 'u';
       ELSE
              operation := 'd';
       END IF;
       INSERT INTO log_table VALUES (operation, SYSDATE(), USER());
END trg_name;
```

Проектирование баз данных.

> Имея в виду какое-либо предприятие, помысли, точно ли оно тебе удастся.
>
> *К. Прутков Афоризмы №54 Плоды раздумья, не включавшиеся в собрание сочинений*

Наиболее общее определение, для процесса проектирования, по-видимому, может выглядеть примерно так: *проектирование* — это поиск способа удовлетворения функциональных требований средствами, имеющейся технологии с учетом заданных ограничений.

Стадии развития проекта.

Любой программный проект за время своей жизни проходит несколько стадий. В данном разделе мы кратко рассмотрим только основные этапы.

Разработка стратегии.

Данный этап не относится к собственно жизненному циклу проекта, но именно на этом этапе определяются требования к будущему продукту, определяется объем работ, график их выполнения, затраты и прибыли.

Крайне важным результатом этого этапа являются явно указанные рамки проекта. Т. е. указание, что **не** будет реализовано данным проектом.

В случае если результаты данного этапа удовлетворяют как заказчика, так и разработчика, происходит переход к разработке продукта.

Анализ.

Этап анализа — это подробное исследование бизнес-процессов (*функций*) и информации, необходимой для их выполнения (*сущностей* с их *атрибутами* и *отношениями*)

Аналитики собирают и фиксируют информацию в двух разных, но взаимосвязанных формах, которые в результате представляются в виде

иерархии функций,
: которая содержит информацию о событиях и процессах, происходящих в бизнесе, а также их взаимосвязи;

модели "сущности — отношения",
: которая описывает ключевые понятия прикладной области, взаимосвязи между этими понятиями и информацию их атрибуты.

На этом же этапе необходимо зафиксировать *бизнес-правила* — ограничения на поведение системы. Например, максимальный размер скидки покупателю.

Проектирование.

На основе результатов анализа при проектировании применительно к базам данных разрабатываются:

- *схема базы данных* на основании модели сущностей, и
- *спецификация модулей* на основе иерархии функций.

Схема базы данных содержит описание всех объектов базы: таблиц, индексов, представлений и т. д. Спецификация модуля содержит название модуля, язык реализации,

условную сложность, интерфейс вызова, используемые таблицы и описание.

Результаты этапа оформляются в единый документ, который принято называть *технической спецификацией*.

Реализация и отладка.

На этом этапе создаются программы, определённые на этапе проектирования. В результате получаются *исходные тексты*, *программы и объектные модули*, а также *тесты программ*.

На всех этапах разработки, а в особенности на этапах анализа и проектирования крайне важно строить *инфологическую модель*, иначе говоря "словесный портрет" желаемого результата. Это позволяет провести предварительную формализацию. Кроме того, в процессе проектирования и анализа должно классифицировать планируемые функции по степени важности — проводить, так называемый, Moscow-анализ:

Must have —
обязательные функции;

Should have —
желательные функции;

Could have —
возможные функции;

Won't have —
отсутствующие функции.

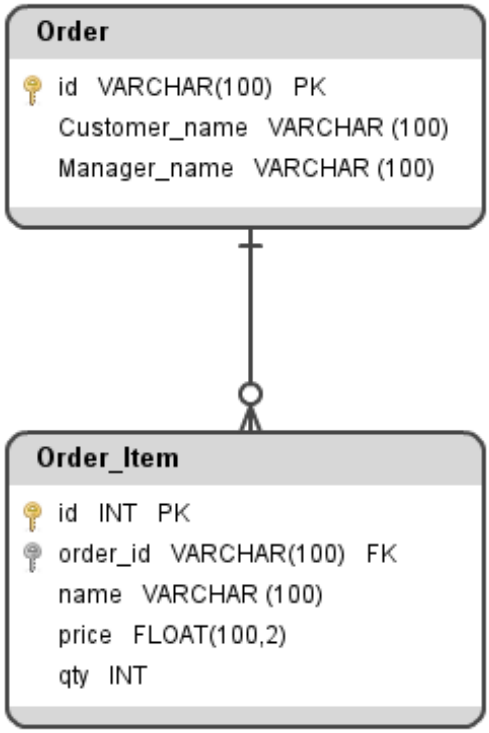
Особенно важны первая и последняя категории, т. к. первая содержит все критические функции, а последняя честно объявляет, какие функции реализованы не будут. Реализация второй и третьей категории зависит от временных и финансовых рамок.

Проектирование базы данных.

На этапе проектирования, применительно к базам данных выполняются следующие задачи.

1. Построение нормализованной информационной модели, реализуемой в рамках базы данных. Такая модель должна быть приведена как минимум к третьей нормальной форме.
2. Построение логической и физической модели данных.
   При этом необходимо выявить нереализуемые конструкции в модели "сущность — отношение" и в определениях сущностей; выделить супертипы и подтипы сущностей; изучить все первичные и внешние ключи; провести денормализацию; если позволяет СУБД, выделить хранимые процедуры и функции; выделить ограничения целостности данных; спроектировать триггеры для всех бизнес-правил и правил целостности данных, которые не могут быть реализованы как ограничения; разработать стратегию индексирования и кластеризации таблиц (если позволяет СУБД); провести оценку размеров всех таблиц, кластеров и индексов; определить уровни доступа к данным; для распределенных баз данных разработать сетевую топологию и механизмы доступа к удаленным данным.
3. Создание базы данных для разработки.
   Для того, чтобы отстающие разработчики могли спокойно заканчивать свою работу, рекомендуется создавать несколько баз данных, которые циклически используются в процессе разработки. Создание таких баз данных при отсутствии администратора базы данных должно проводиться проектировщиком.

ER-диаграммы.

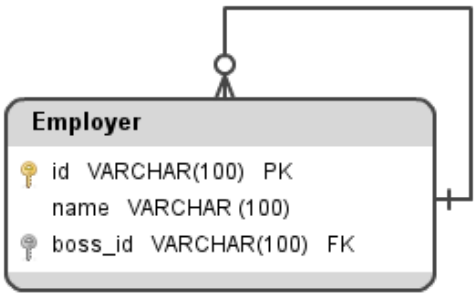


Наиболее частым представлением модели "сущность-отношение" является так называемая *ER-диаграмма*. В настоящий момент наибольшее распространение получили: нотация Питера Чена, Crow's Foot (воронья лапа) и UML. Ниже будет представлен один из её вариантов (Crow's Foot), который, с точки зрения авторов, является наиболее удачным, т. к. одинаково хорошо применим как к построению диаграммы на компьютере, так и на бумаге[20].

На таких диаграммах сущности, которым соответствуют таблицы в базе данных, отображаются в виде прямоугольников, содержащих имя таблицы, первичный ключ и атрибуты. Различного рода отношения и взаимозависимости между сущностями на ER-диаграммах отражаются в виде линий или дуг.

На начальном этапе рекомендуется отображать только название сущности.

Имя сущности дается в единственном числе. При рисовании от руки атрибуты первичного ключа подчеркиваются, а на компьютере отмечаются тем или иным способом (в данном случае золотистый ключ перед первыми атрибутами и пометка PK после указания типа атрибута). Прямая от сущности «Позиция в счете» к сущности «Счет» символизирует, что счет включает в себя (состоит из) отдельные позиции. "Птичья лапка" со стороны сущности «Позиция в счете» означает, что в счет может включаться несколько позиций. Кружок,— что в счете вообще может не быть позиций (например, только что созданный счет). Отсутствие "птичьей лапки" со стороны сущности «Счет» означает, что позиция счета всегда принадлежит только одному счету. Вертикальная черта — позиция счета **обязательно** принадлежит какому-нибудь счету, т. е. позиция по счету не может существовать сама по себе.



При первоначальной прорисовке ER-диаграмм могут возникать «ненормальные» связи между отношениями. Под ненормальностью понимается либо невозможность создания таких связей в реляционной СУБД, либо возможность возникновения проблем при последующей работе с данными.

Один из возможных вариантов — «свиное ухо». Т. е. таблица ссылается сама на себя. Как правило, так выглядит древовидная структура перенесённая в реляционную модель. Таблицу создать не проблема, но как внести первую запись? И что делать, если ее надо удалить?

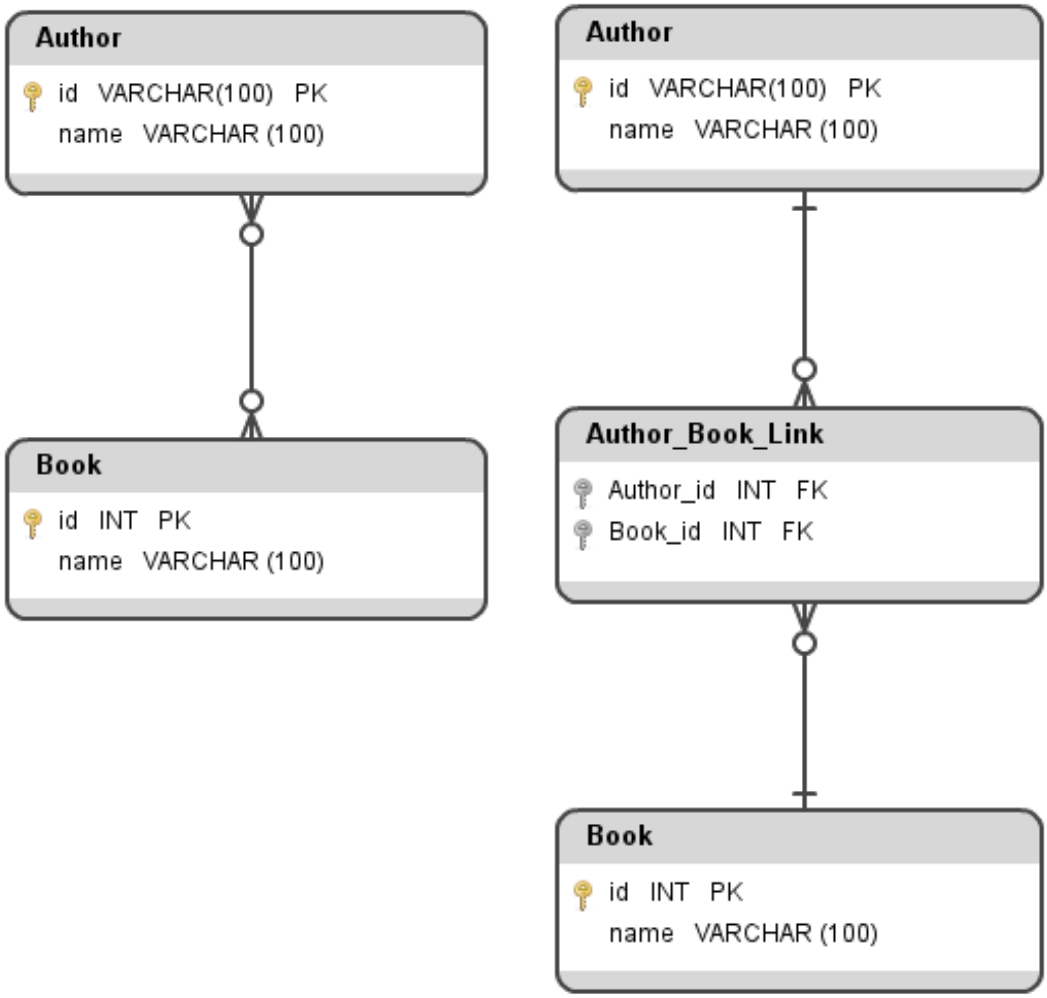
Есть два пути:

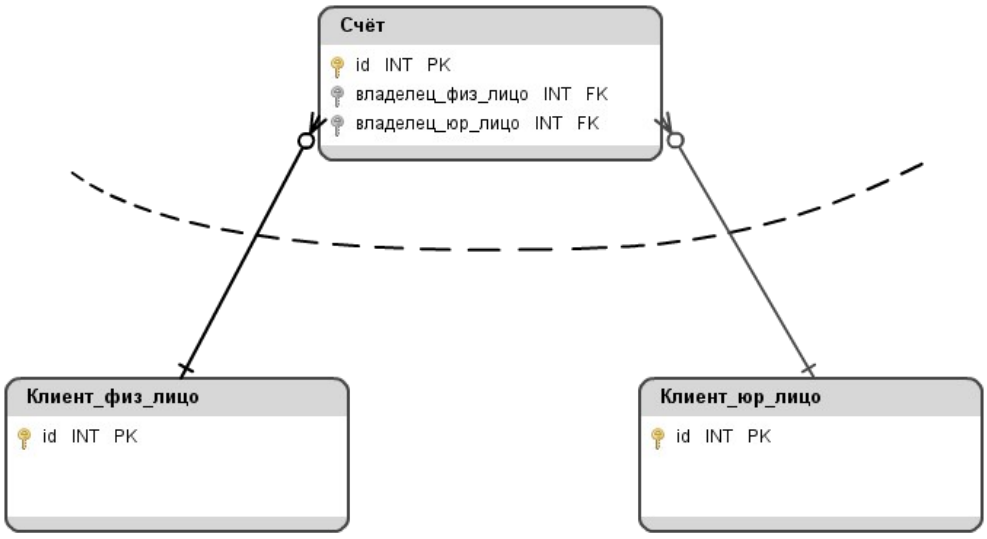
- отключить ссылочное ограничение целостности, внести фиктивную запись и включить ссылочное ограничение целостности. В дальнейшем корень дерева подвязываем к этой фиктивной записи. Проблема в том, что есть фиктивная запись, которую надо учитывать в логике программ.
- отключить ссылочное ограничение целостности, внести первую запись и включить ссылочное ограничение целостности. Проблема в том, что при модификации этой

20 Гениальные озарения случаются в самых неожиданных местах и, как правило, не на работе.

записи ссылочную целостность опять надо будет выключать/включать.

Следующие два рисунка демонстрируют не реализуемую, в рамках реляционных СУБД, связь «многие-ко-многим», и способ разрешения этой ненормальной связи.

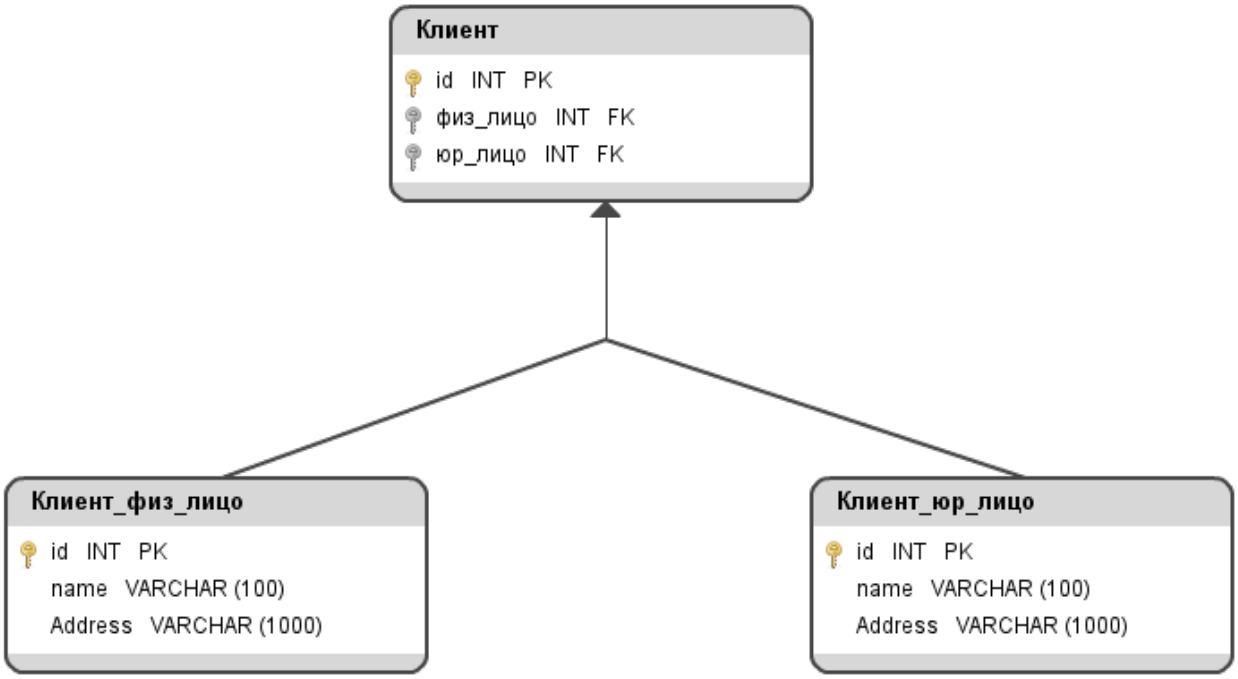




Третий вариант ненормальной связи — дуга, обозначающая, что реализуется только один вариант: либо счет принадлежит юридическому лицу, либо физическому лицу. Как правило, наличие дуги означает, что в этом месте диаграммы

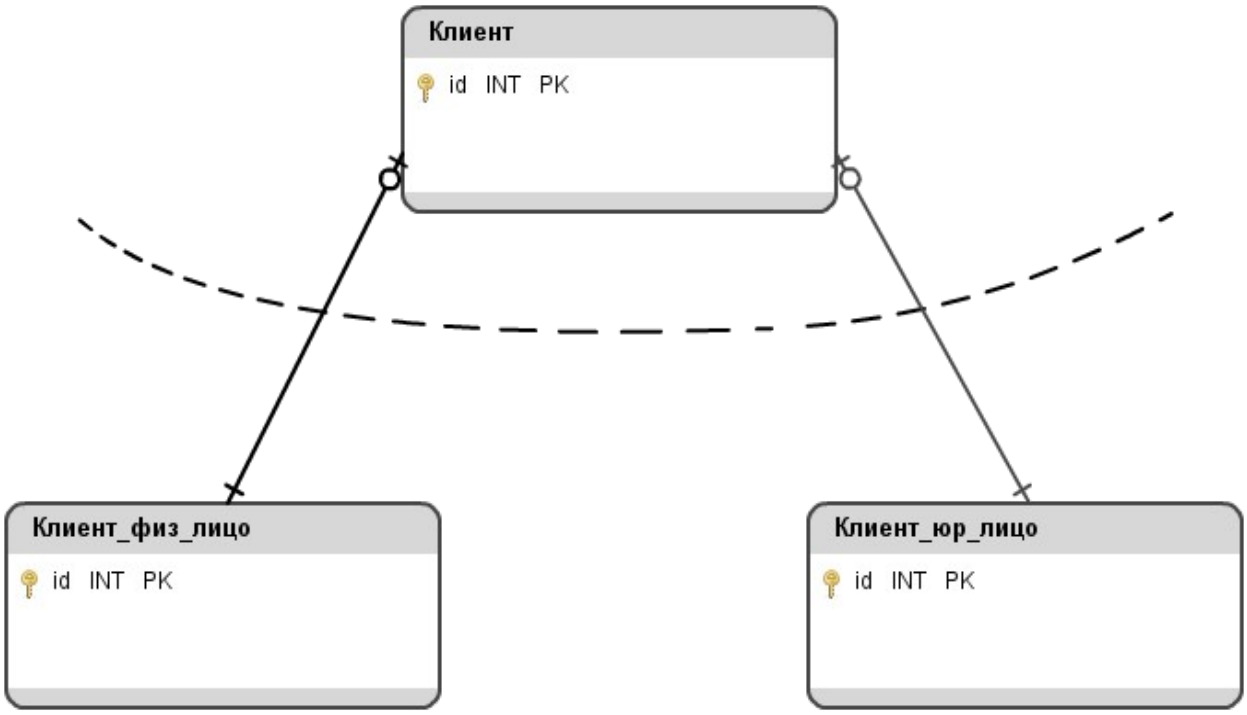
должно появиться наследование сущностей (в терминах ООП).

Т.е. либо счет должен иметь наследников: счет_юру_лица, счет_физ_лица, каждый из которых принадлежит соответствующему клиенту. Либо счет должен принадлежать обобщенному клиенту, от которого наследуются клиент_юру_лицо и клиент_физ_лицо.



Это диаграмма наследования в терминах ООП, т. е. общие атрибуты двух отношений выносятся в родительскую сущность. Некоторые СУБД, например PostgreSQL, позволяют реализовать ее явным образом (CREATE TABLE … INHERITS).

Если СУБД не позволяет реализовать наследование явным образом его можно эмулировать через дугу. Обратите внимание, что связи между наследниками и родителем отличаются от того, что было показано ранее. Если дуга перечеркивает связи вида «один-к-одному», то это «нормальная» дуга. Если есть связи типа 1:n — ненормальная.



Нормализация.

Аномалии работы с данными.

Представим себе, что у нас есть таблица следующего вида, описывающее учебные курсы:

№ зачётной книжки	ФИО студента	Преподаватель	Дисциплина
765765	Иванов Я. Я.	Кодд	Базы данных
765765	Иванов Я. Я.	Буч	ООПиД
765765	Иванов Я. Я.	Стивенс	Сети TCP/IP
765321	Петров А. А.	Кодд	Базы данных
765321	Петров А. А.	Стивенс	Сети TCP/IP

Допустим также, что г-н Иванов. не понимая собственного счастья, отказывается посещать курс ООПиД. При удалении соответствующей строки из таблицы, мы удаляем единственное упоминание о данном курсе и его преподавателе! Можно представить, что г-н Петров, напротив, изъявил желание посещать ООПиД, и мы внесли дополнительную строку в таблицу. Проблема только в том, что при внесении мы допустили ошибки, и строка получилась такой:

765321	Петрова А.	Бутч	ООПиД

Тем самым, сами того не желая, мы „создали” нового студента и нового преподавателя. Если предположить, что „ООПиД” факультативный курс, а „Базы данных” и „Сети TCP/IP” — обязательные, то можно обнаружить ещё один тип проблем: достаточно представить, что мы должны внести информацию о новом студенте, но добавили информацию о посещении только одного обязательного курса.

Столь же легко, можно показать проблемы, которые возникают при модификации данных.

Все эти проблемы называют *аномалиями удаления, вставки и модификации данных*,

соответственно.

По-сути, единственной проблемой показанной выше таблицы является то, что одна и та же информация хранится в нескольких местах, т. е. имеется избыточность. Напротив, если следовать лозунгу „один факт — в одном месте”, можно смело рассчитывать, что показанных выше аномалий наблюдаться не будет.

Неизбежные определения.

Прежде чем переходить к вопросам нормализации отношений, дадим несколько ключевых определений.

Принято говорить, что множество **B** *функционально зависит* от множества **A**, если любое значение из **A** определяет одно значение из **B** (Это же можно сказать и в обратную сторону: **A** *функционально определяет* **B**). Аналогично, **B** *многозначно зависит* от **A**, если значение из **A** определяет набор значений из **B**.

C *транзитивно зависит* от **A**, если **B** функционально зависит от **A** и функционально определяет **C**, но при этом не существует функциональной зависимости **A** от **B**.

Декомпозиция — это совокупность отношений-проекций исходного отношения, а также процесс получения этих проекций.

Полная декомпозиция — такая совокупность таблиц-проекций, что их естественное соединение полностью совпадает с исходным отношением[21].

Под нормализацией понимают процесс приведения набора таблиц к *нормальным формам*, которые рассматриваются ниже.

Таблица находится в *первой нормальной форме* (1НФ), если все ее атрибуты содержат атомарные значения[22].

Таблица находится во *второй нормальной форме* (2НФ), если она находится в 1НФ, и *все неключевые атрибуты функционально полно зависят от первичного ключа* (т. е. от всего ключа в целом).

Таблица находится в *третьей нормальной форме* (3НФ), если она находится в 2НФ, и *все неключевые атрибуты зависят только от первичного ключа*.

Таблица находится в *нормальной форме Бойса-Кодда* (НФБК), если в ней *нет транзитивных зависимостей*.

Существует два возможных варианта:

1. неключевой элемент функционально зависит от атрибута или нескольких атрибутов, входящих в составной ключ (это критерий нарушения 2НФ);
2. атрибут, входящий в составной ключ, зависит от части первичного ключа.

Имеется альтернативная формулировка критерия нормальной формы Бойса-Кодда: *таблица находится в НФБК тогда и только тогда, когда детерминанты функциональных зависимостей являются первичными ключами.*

Таблица находится в *четвертой нормальной форме* (4НФ), если она находится в НФБК, и *не содержит многозначных зависимостей*.

Такие зависимости возникают в связи с многозначными атрибутами, как следствие, для решения проблемы необходимо вынести каждый многозначный атрибут в отдельную таблицу вместе с ключом, от которого он зависит.

Таблица находится в *пятой нормальной форме* (5НФ), *если в каждой ее полной декомпозиции все проекции содержат возможный ключ.* Таблица, не имеющая ни одной полной декомпозиции также находится в 5НФ. (Интересно отметить, что 4НФ представляет из себя полную декомпозицию из двух проекций.)

21 Разумеется, это должно быть справедливо для **любого** набора кортежей исходного отношения.

22 Это означает, что в реляционной базе данных все таблицы находятся в 1НФ, по определению

Процесс нормализации.

Как уже было сказано, отношение в реляционной модели всегда находится в 1НФ и никаких дополнительных усилий для приведения отношения к 1НФ не требуется.

Возьмём уже рассмотренную таблицу и добавим в неё ещё один столбец, с указанием рекомендованной литературы:

№ зачётной книжки	ФИО студента	Преподаватель	Дисциплина	Литература	Часы	Оценка
765765	Иванов Я. Я.	Кодд	Базы данных	Edgar Frank Codd. „A Relational Model of Data for Large Shared Data Banks"	72	
765765	Иванов Я. Я.	Кодд	Базы данных	Codd, E.F.; Codd S.B. and Salley C.T. (1993). „Providing OLAP to User-Analysts: An IT Mandate"	72	
765765	Иванов Я. Я.	Буч	ООПиД	Booch G. „Object-Oriented Analysis and Design with Applications (3rd Edition)"	48	
765765	Иванов Я. Я.	Стивенс	Сети TCP/IP	W. Richard Stevens. „TCP/IP Illustrated, Vol. 1: The Protocols"	36	
765321	Петров А. А.	Кодд	Базы данных	Edgar Frank Codd. „A Relational Model of Data for Large Shared Data Banks"	72	
765321	Петров А. А.	Кодд	Базы данных	Codd, E.F.; Codd S.B. and Salley C.T. (1993). „Providing OLAP to User-Analysts: An IT Mandate"	72	
765321	Петров А. А.	Стивенс	Сети TCP/IP	W. Richard Stevens. „TCP/IP Illustrated, Vol. 1: The Protocols"	36	

Очевидно, что в данной таблице существует целый ряд функциональных зависимостей. Вот только часть из них:

(№ зачётной книжки) → (ФИО студента)

(Дисциплина) → (Преподаватель)

(Дисциплина) → (Часы)

(№ зачётной книжки, Дисциплина) → (Оценка)

Необходимо понимать, что в контексте нормализации отношений, понятие функциональной зависимости носит ярко выраженный семантический характер, и определяется предметной областью. Т. е. нельзя глядя на таблицу сказать, может ли преподаватель вести несколько предметов, невозможно понять от чего зависит список рекомендованной литературы (т. е. он может ли он зависеть от преподавателя, или же он зависит исключительно от изучаемой дисциплины) и т. п. Напротив, всё это определяется требованиями заказчика.

С практической точки зрения, есть два пути приведения указанного отношения к НФБК:

1. последовательно проводить декомпозицию отношений и проверять критерии нормальных форм;
2. сразу обратить внимание на альтернативную формулировку НФБК и заметить, что она не имеет ссылок на нормальные формы низших порядков!

Итак,

а) все ФЗ в исходном отношении нам известны,

b) для большей наглядности критерий НФБК можно записать так: if $A \rightarrow B \Rightarrow A$ is PK

Из этого получаем список первичных ключей для будущих нормализованных отношений: № зачётной книжки, Дисциплина, (№ зачётной книжки, Дисциплина)

Проводя декомпозицию получим следующий набор таблиц:

№ зачётной книжки	ФИО студента
765765	Иванов Я. Я.
765321	Петров А. А.

Преподаватель	Дисциплина	Часы
Кодд	Базы данных	72
Буч	ООПиД	48
Стивенс	Сети TCP/IP	36

№ зачётной книжки	Дисциплина	Оценка
765765	Базы данных	
765765	ООПиД	
765765	Сети TCP/IP	

Нормальные формы более высокого порядка решают проблемы связанные с использованием составных первичных ключей, в которых лишь часть ключа сама по себе содержит информацию. Любая таблица в 3НФ без составного первичного ключа автоматически находится в 5НФ.

Преподаватель	Дисциплина	Литература
Кодд	Базы данных	Edgar Frank Codd. „A Relational Model of Data for Large Shared Data Banks"
Кодд	Базы данных	Codd, E.F.; Codd S.B. and Salley C.T. (1993). „Providing OLAP to User-Analysts: An IT Mandate"
Буч	ООПиД	Booch G. „Object-Oriented Analysis and Design with Applications (3rd Edition)"
Стивенс	Сети TCP/IP	W. Richard Stevens. „TCP/IP Illustrated, Vol. 1: The Protocols"
Кодд	Базы данных	Edgar Frank Codd. „A Relational Model of Data for Large Shared Data Banks"
Кодд	Базы данных	Codd, E.F.; Codd S.B. and Salley C.T. (1993). „Providing OLAP to User-Analysts: An IT Mandate"
Стивенс	Сети TCP/IP	W. Richard Stevens. „TCP/IP Illustrated, Vol. 1: The Protocols"

Рассмотрим указанное отношение. Здесь нет функциональных зависимостей, но есть две многозначных зависимости:

- Дисциплина ->> Литература
- Дисциплина ->> Преподаватель

Проблема состоит в том, что если появляется новый преподаватель, мы вынуждены вставлять несколько строк с указанием нового преподавателя и всем списком рекомендованной литературы. Аналогично, и для новой книги в списке. Иными словами, здесь нарушается критерий 4НФ.

Решение состоит в том, чтобы разнести многозначные зависимости по отдельным отношениям.

Денормализация.

После успешно проведенной нормализации проектировщик должен проанализировать получившуюся структуру таблиц на необходимость *денормализации*, т. е. внесения

избыточности в существующие таблицы. Данный шаг может показаться странным и неправильным. Однако, он может быть вполне оправдан, если внесение избыточной информации положительно сказывается на общей производительности системы.

Например, естественной структурой, описывающей заказ в магазине, будет структура из трех таблиц: "заказ", "товар" и "элементы заказа". Первая таблица содержит общую информацию о заказе. Вторая — информацию о наименовании товара, цене и т. п. Третья — содержит информацию о номере заказа, товаре и количестве данного товара в данном заказе. Очевидно, что при частом поиске заказов с общей суммой заказа свыше определенного числа возникают существенные накладные расходы на связывание этих трех таблиц.

В качестве альтернативного решения можно завести дополнительную колонку в таблице "заказ", содержащую полную сумму заказа. Обновление значений в данной колонке должно происходить либо посредством триггера при изменении состава заказа, либо периодическим выполнением соответствующей команды UPDATE.

Альтернатива процессу нормализации.

Основная проблема классического подхода состоит в том, что он чрезвычайно трудоёмкий. Его трудоёмкость определяется несколькими причинами:

- неочевидность наличия функциональной или многозначной зависимости между атрибутами отношения.
- отсутствие формальных критериев, которые позволяли бы определить, что является сущностью.

Тарасов в своей работе „Метод проектирования логической структуры реляционной БД без нормализации таблиц” представил альтернативный подход к проектированию схемы базы данных, позволяющий получить более качественную структуру базы данных с меньшими трудозатратами.

Суть метода сводится к введению понятия функционального требования, как детерминанта поведения системы (программного обеспечения) на заданное событие. На основе функциональных требований возможно дать формальное определение понятиям „сущность” и „атрибут”. Что позволяет автоматически определить набор таблиц в базе данных и набор атрибутов для каждой из этих таблиц. Там же показано, что в созданных по такому методу таблицах не будет аномалий работы с данными.

Выбор первичных ключей.

Следующим шагом, как правило, определяют первичные ключи. Несмотря на кажущуюся простоту этой операции, проектировщик должен убедиться, что выбранная кандидатура с одной стороны всегда будет уникально идентифицировать строку таблицы[23], а с другой — будет неизменной в течении всей жизни данной строки[24]. На этом же этапе необходимо принять решение об использовании суррогатных первичных ключей. Введение таких ключей нарушает критерий 2НФ, и, что гораздо хуже, потенциально позволяет создание записей в таблице с повторяющимся «истинным» первичным ключом.

23 Например, нельзя брать за первичный ключ ФИО сотрудников организации

24 Можно ли взять за первичный ключ номер кузова, шасси или двигателя автомобиля?

Хранилища данных.

Появление систем хранения данных (*СХД, хранилище данных, Data Warehouse, DWH*) является, прежде всего, естественным эволюционным процессом. Для понимания этого утверждения давайте представим, что в наличии имеется реляционная СУБД[25], которая обеспечивает работу системы электронной продажи билетов. Еще одним важным элементом нашего рассмотрения будет условие не уменьшения объёма данных в нашей базе, т. е. данные в БД вносятся, но не удаляются.

Вполне очевидно, что спустя достаточно короткое время в системе будет накоплен значительный объем данных, статистический анализ которых позволит нам найти новые скрытые закономерности, существующие в нашей прикладной области[26]. Чем больший объем данных затрагивается при анализе, тем выше точность получаемых результатов, и тем больший практический интерес они представляют. В настоящее время такие сверхбольшие базы данных получили название *хранилища данных* (data warehouse). Их типичный размер — тера- и петабайты. Характерной особенностью хранилищ данных является тот факт, что данные в хранилище вносятся на регулярной основе и, практически, никогда не модифицируются и не удаляются. Более того, частота и объем вносимых данных крайне малы по сравнению с частотой запросов выборки данных и объемом данных, которые затрагиваются этими запросами.

Рассмотрение алгоритмов такого статистического анализа выходит за рамки этой работы. Интересующиеся читатели могут самостоятельно поднять литературу по теме систем поддержки принятия решений, либо ``спросить" у google про business intelligence и data mining.

Рассматриваемая нами база данных обязана быть сильно нормализованной, т. к. это позволяет решить множество проблем, связанных с аномалиями при работе с данными. С другой стороны, аналитические запросы характеризуются крайне высокой степенью связности по отношению к нормализованным таблицам нашей базы. Учитывая, что вычисление соединения весьма дорогая операция в плане потребляемых ресурсов, можно с уверенностью утверждать, что аналитические запросы к базе данных будут занимать существенное время и потреблять массу системных ресурсов.

Итак, на лицо наличие определенного противоречия: с одной стороны нам необходимо использовать высоко нормализованную схему базы данных, чтобы соответствовать парадигме «один факт в одном месте», а, с другой стороны, это ведет к снижению производительности системы. Вполне очевидно и решение данной проблемы. А именно, необходимо разделить оперативную и аналитическую обработку информации по разным базам данных.

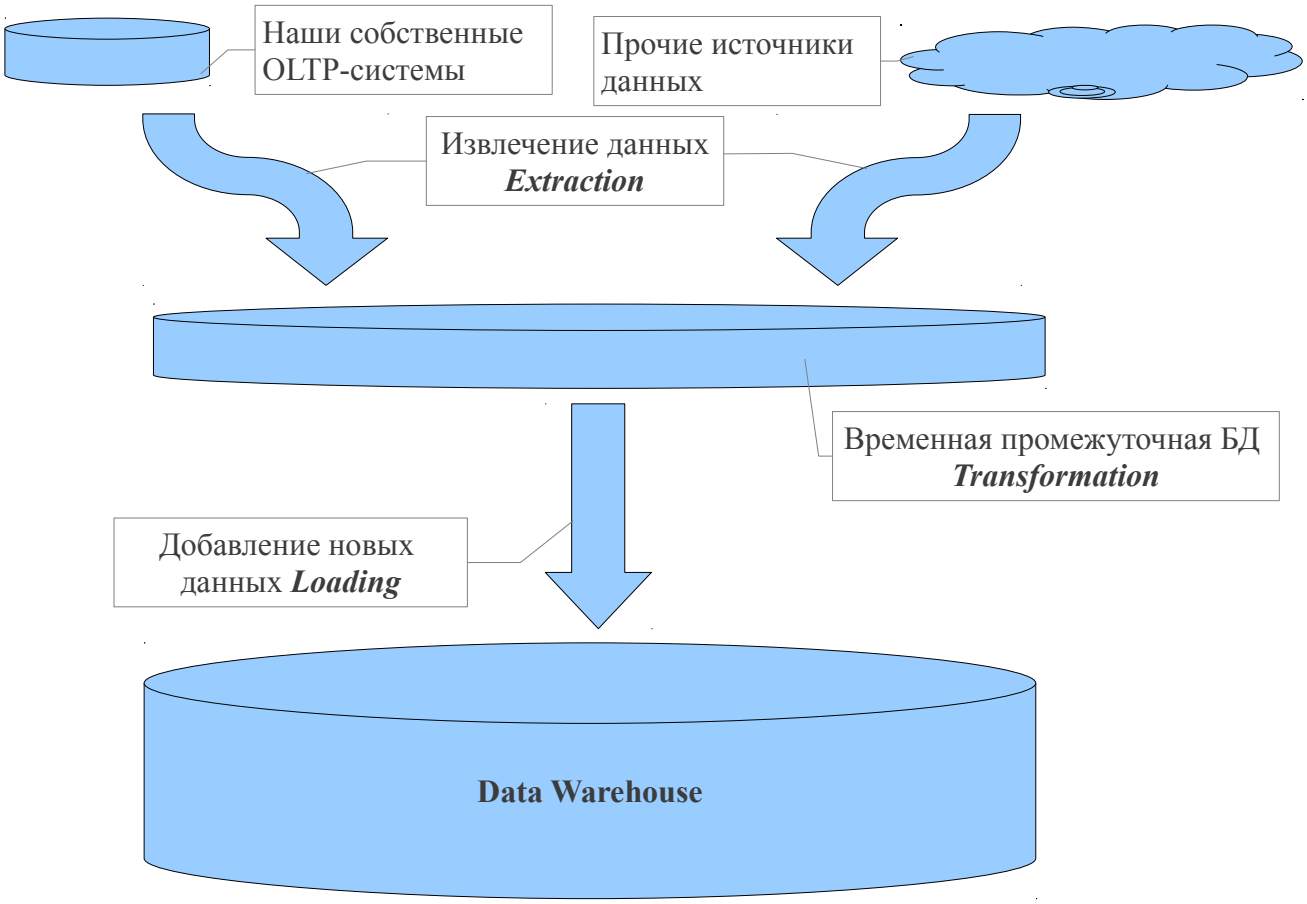
Такое разделение позволит не только разнести нагрузку по разным системам, но и оптимизировать структуру таблиц[27] под работу с аналитическими запросами. Более того, это позволит нам использовать различные по своей природе источники данных для получения дополнительной информации и ее последующего хранения в DWH.

Архитектура типичного DWH представлена на следующем рисунке.

25 Вообще говоря, реляционная природа СУБД не обязательна, но для большей наглядности изложения будем считать её таковой.

26 Как минимум, у нас есть надежда на это.

27 Существует утверждение: «если есть проблемы с производительностью программы, необходимо проверить структуру представления данных — скорее всего, она неправильна.»



Проблема производительности не единственная, с которой можно столкнуться при отсутствии DWH. Основной и наиболее важной проблемой можно назвать низкую достоверность и несогласованность исходных данных, и, как следствие, получаемых результатов, при выполнении аналитических исследований.

Например, если одно подразделение организации создаёт аналитический отчёт по средам и использует для его создания некоторое множество источников информации, а другое подразделение создаёт аналогичный отчёт по пятницам на основе другого множества источников информации, то будет неудивительно, если результаты анализа будут отличаться. Можно назвать, как минимум, три основных причины для этого: различные источники данных, различные расписания создания отчётов и различные алгоритмы проведения анализа.

Наличие DWH позволяет организовать единый консолидированный источник данных для анализа. Имея подобный источник можно стандартизовать алгоритмы, используемые при создании отчётов, и расписание их создания.

Ещё одна проблема — невозможность перейти от данных к информации. Например, SEO предприятия может задаться вопросом: «Как отличается текущая активность пользователей сайта, от их активности месяц назад, два месяца назад и т. д. на протяжении года?» При наличии, например, только журналов работы http–сервера за указанный промежуток времени, получение ответа на данный вопрос возможно, но сопряжено с существенными трудностями. При наличии DWH, содержащего информацию о работе сайта, получение ответа сводится к формированию соответствующего запроса.

Процесс переноса данных из источников информации в DWH называется ETL — по трём стадиям, которые принято выделять в ходе такого переноса:

1. Извлечение данных (Extraction) — на этой стадии данные извлекаются из источников информации и помещаются в промежуточную БД. Программы, извлекающие данные из источников, индивидуальны для каждого источника. Это может быть робот, обходящий сайты и выбирающий требуемую информацию, SQL-клиент, выбирающий информацию из OLTP-системы, и т. д.
2. Преобразование (Transformation) — на этой стадии извлечённые данные преобразуются к виду, в котором они будут храниться в DWH. Как указано на рисунке «Error: Reference source not found», для выполнения операций по преобразованию данных используется промежуточная БД.
3. Загрузка (Loading) — на этом этапе происходит перенос подготовленных данных в DWH. Реализация данного этапа зависит от конкретной СУБД, используемой для построения DWH. Например, Oracle имеет механизм transportable tablespace, используя который, перенос данных в DWH сводится к копированию файлов данных на DWH-сервер и подключению их к СУБД.

Информация, хранящаяся в DWH имеет крайне высокую ценность, т. к. охватывает абсолютно все аспекты деятельности организации. Таким образом, возникают две ключевые задачи в плане обеспечения функционирования DWH:

- в ходе ETL-процесса на первых двух стадиях необходимо проверять новые данные на предмет целостности и непротиворечивости, т. е. данные должны проходить определенную очистку от «мусора»;
- доступ к данным, хранящимся в DWH, должен быть строго регламентирован.

Последний пункт требует отдельного рассмотрения. Напомним, DWH является единственным источником информации при принятии решений в рамках организации. Как следствие, либо весь персонал, имеющий доступ к данным, должен входить в список лиц имеющих допуск к ключевой информации, либо, что много правильней, DWH должен предоставлять механизмы разграничения доступа к данным. Вполне очевидно, что информация, требующаяся двум различным подразделениям, будет различна. Одним из возможных решений задачи по разграничению доступа к данным является создание небольших DWH, функционирующих в интересах конкретного подразделения или малого числа подразделений. Данные DWH принято называть киосками либо витринами данных (Data Marts, DM). Основные отличия DM от «настоящих» DWH состоит в том, что: а) единственным источником данных для DM служит основное DWH, и б) информация, содержащаяся в DM, является тематически ориентированной. Дополнительным плюсом от использования DM является снижение нагрузки на основное DWH.

Четырехуровневая структура данных, возникающая при использовании DM, является на сегодняшний день устоявшимся «классическим» решением в области построения хранилищ данных:

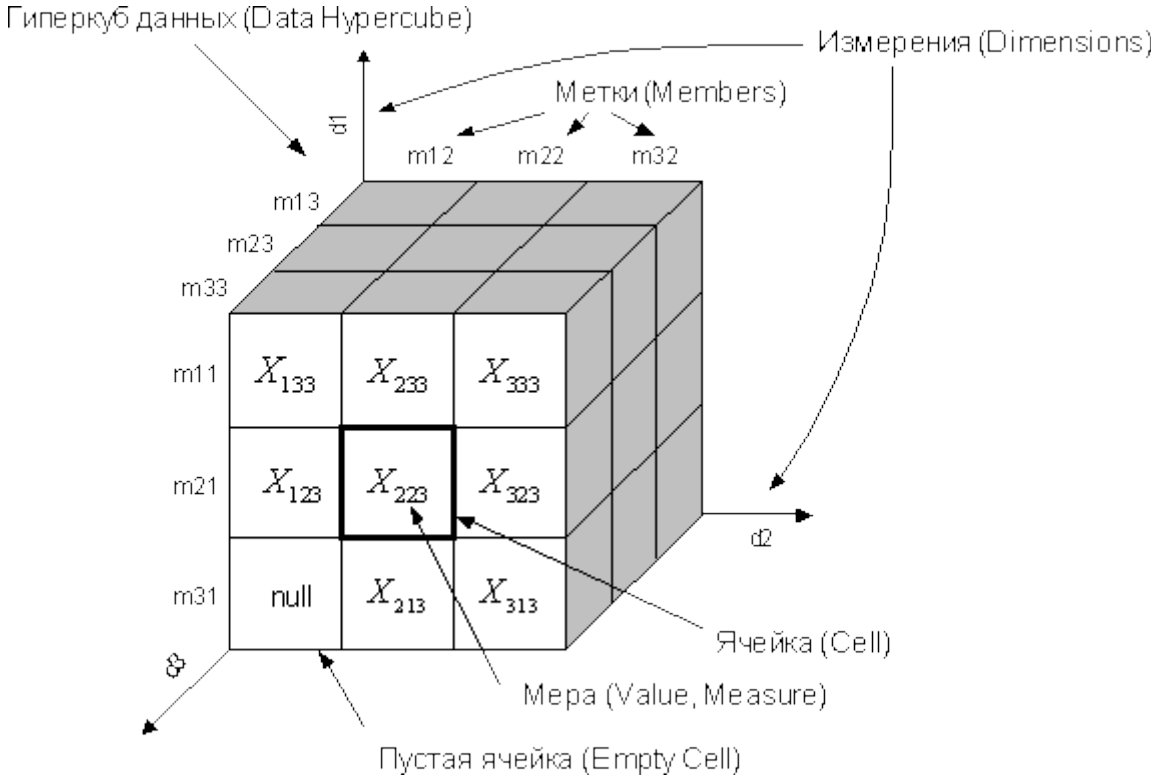
- первый уровень, представленный OLTP-системами, хранит только оперативные данные (например: текущий баланс счета, текущее состояние проданных билетов на конкретный рейс и т. п.);
- второй уровень данных представлен корпоративным хранилищем данных, и содержит наиболее детализированные данные исторического характера: кредитная история клиента банка, результаты переписей населения и т. п.;
- третий уровень данных представлен витринами данных, и содержит агрегированные данные, которые относятся к отдельной тематически ориентированной области: численность населения отдельных регионов в конкретные годы;
- четвёртый уровень представлен данными, расположенными на компьютерах пользователей. Это временные и наиболее агрегированные данные, которые интересны данному конкретному аналитику. Например, изменение (прирост и убыль) населения страны за определенный исторический период.

При построении DWH необходимо учитывать, что цикл разработки для хранилища данных является обратным по отношению к классическим программным продуктам. Если разработка классической программы выглядит как последовательность следующих этапов:

1. сбор требований
2. анализ
3. проектирование
4. собственно программирование
5. тестирование и отладка
6. введение в эксплуатацию,

то для DWH будет характерна обратная последовательность:

1. организация хранилища данных
2. сбор и объединение данных
3. выявление тенденций
4. написание программ, опираясь на данные из хранилища
5. проектирование СППР
6. анализ получаемых результатов
7. определение требований

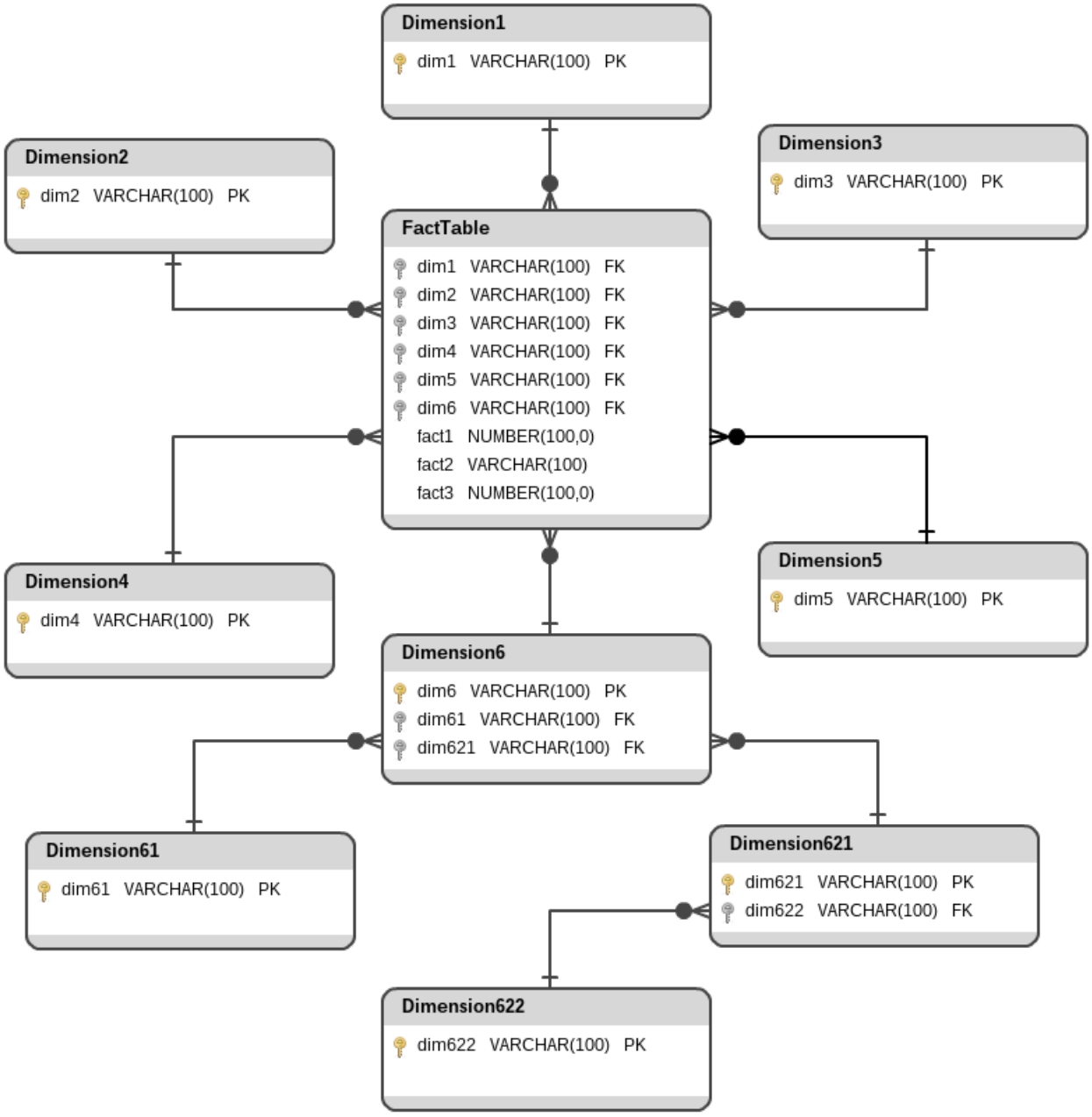


Данное отличие объясняются в первую очередь тем, что применительно к хранилищам данных во главу угла ставятся именно данные, т. к. их характерное время жизни в DWH исчисляется десятилетиями, что существенно превышает срок жизни практически всех алгоритмов, применяемых при обработке этих данных.

Вернёмся к нашей задаче по переходу от OLTP-системы к OLAP. Чтобы эффективно организовать хранилище данных, необходимо понять, в каком виде данные должны быть представлены на прикладном уровне. Иными словами, надо понять, как их себе представляет аналитик. Оказывается, что аналитики оперируют данными, представленными в виде гиперкубов[28]. Вдоль ребер такого гиперкуба откладываются величины, имеющие семантику

28 Строго говоря, эти объекты не являются гиперкубами в математическом смысле этого слова, однако, данный термин является устоявшимся и мы будем использовать именно его.

размерности: время продажи, географическое расположение места продажи и т. п. В ячейку гиперкуба с конкретными значениями размерностей вносятся величины, имеющие семантику факта (измеряемой величины): стоимость билета, класс обслуживания и т. п. Простым и наглядным примером двумерного гиперкуба является страница с четвертными и годовыми оценками в дневнике школьника: по одной из размерностей указаны наименования предметов, по другой — временны*е* интервалы, за которые выставляется оценка. В нужной ячейке гиперкуба вносится значение факта — оценка по соответствующему предмету за соответствующую четверть. Таким образом, для успешной организации хранилища данных необходимо отобразить гиперкубы предметной области на структуры, хранящиеся в СУБД.



Для реляционных СУБД такими структурами являются таблицы, но алгоритм построения схемы базы данных для DWH кардинально отличается от того, что используется в случае с OLTP системами. Прежде всего, в предметной области определяется набор гиперкубов, которые будут хранить наиболее детализированную информацию. Каждый из таких гиперкубов отображается на таблицу фактов (fact table) со следующей структурой:

каждый элемент размерностей гиперкуба и каждый отдельный факт отображается на соответствующую колонку в таблице фактов. Т. е. табельная страница школьного дневника будет отображаться на таблицу из трёх колонок: название дисциплины, номер четверти и оценка. Аналогично, если бы некий гиперкуб имел бы четыре размерности: долгота, широта, высота над уровнем моря, дата и время, а в каждой ячейке содержалось бы пять значений фактов (например: температура воздуха, влажность воздуха, атмосферное давление, скорость ветра, направление ветра), то результирующая таблица фактов имела бы девять колонок для каждого из указанных параметров.

Наличие в таблице фактов только допустимых значений размерностей контролируется ссылочной целостностью между таблицей фактов и соответствующими таблицами размерностей (dimension tables).

Соответствующая структура таблиц на ER-диаграмме представляется в виде таблицы фактов, которая окружена таблицами размерностей, с каждой из которых имеет связь типа 1:n. В случае если та или иная размерность допускает агрегирование, то ER-диаграмма дополняется таблицами размерностей более высокой степени агрегации, причем могут возникать альтернативные пути агрегации: например размерность „дата” может агрегироваться в цепочку размерностей „месяц” – „квартал” – „год”, а может, альтернативно, агрегироваться в „неделю”. Характерный внешний вид описанных ER-диаграмм дал им собственные названия: Star-schema и Snowflake-schema. Данные структуры таблиц являются типовыми для хранилищ данных, построенных на основе реляционных СУБД.

Оптимизация.

Используя термин «оптимизация», в контексте баз данных, необходимо учитывать, что в него может вкладываться три различных, хотя и тесно связанных, понятия:

1. изменение входящего запроса на уровне оптимизатора СУБД, таким образом, чтобы потребление системных ресурсов при его выполнении было минимальным, а результат запроса в точности соответствовал результату оригинального запроса;
2. изменение SQL запроса, программистом или администратором базы данных, с которым клиентское приложение обращается к СУБД, в случаях, когда встроенный оптимизатор СУБД либо не справляется с задачей, либо его работа ведёт к деградации производительности; к этому же пункту можно отнести и ситуации, когда требуется изменить схему базы данных;
3. изменение настроек СУБД, физической конфигурации базы данных и т. п, с целью повысить производительность системы в целом.

Последний пункт выходит за рамки нашего рассмотрения, т. к. входит в круг должностных обязанностей администратора базы данных и высшей степени сильно зависит от используемой СУБД (как от производителя, так и от версии).

Этапы оптимизации запросов в реляционных СУБД

В командах языка программирования SQL отсутствует информация о том, как должен выполняться запрос, и какие методы доступа к данным должны использоваться. Это позволяет СУБД самой выбирать пути выполнения команд, что важно, т. к. почти практически любая команда DML допускает несколько путей исполнения. Рассмотрим следующий пример:

Получим список сотрудников отдела не старше 30 лет с «чистой» зарплатой не менее 50 000 рублей («чистой» называется зарплата после уплаты подоходного налога). Чтобы не задаваться вопросами исчисления эквисоединений, будем считать, что код подразделения нам известен.

```
SELECT e.emp_name, e.salary, e.birth_date
    FROM emp e
    WHERE e.dept_id = 333 AND e.birth_date > SYSDATE - 30*365
        AND salary/1.13>=50000;
```

В этом запросе к таблице emp указаны условия на поля dept_id, birth_date и salary. Если каждое из этих полей индексировано, то запрос может быть исполнен одним из указанных способов:

1. Найти по индексу для dept_id записи, удовлетворяющие первому условию, и проверить для найденных записей второе условие.
2. Найти по индексу для birth_date записи, удовлетворяющие второму условию, и проверить для найденных записей первое условие.
3. Последовательно сканировать таблицу emp и проверять оба условия для каждой записи.

Индексом для salary СУБД пользоваться не может, т. к. это поле используется внутри выражения. (Однако, если данное условие переписать в виде salary>=50000*1.13, индекс будет применим.)

Цель СУБД — выполнить запрос наиболее эффективным способом.

Под оптимизацией понимается построение квазиоптимального[29] процедурного плана выполнения декларативного запроса.

29 план является квазиоптимальным, т. к. нет гарантий, что СУБД для любого запроса выберет наилучший план исполнения запроса. Задача выбора наилучшего плана исполнения решается путем полного перебора всех возможных планов исполнения, что, очевидно, неприемлемо.

План исполнения запроса (EXECUTION PLAN) — это последовательность шагов, каждый из которых либо физически извлекает данные из памяти, либо делает подготовительную работу. Построением этого плана занимается оптимизатор СУБД.

Обработка SQL запроса, поступившего в СУБД, состоит из нескольких этапов:

1. синтаксический анализ;
2. семантический анализ и оптимизация;
3. выбор плана исполнения;
4. создание процедурного плана исполнения;
5. исполнение плана

На первой фазе запрос, представленный на языке SQL, подвергается лексическому и синтаксическому анализу. Лексический анализатор разбивает запрос на лексические единицы – лексемы (наименования полей и таблиц, константы, знаки операций и т.д.). Синтаксический анализатор проверяет синтаксическую правильность запроса. В результате создаётся дерево синтаксического разбора и вырабатывается внутреннее представление запроса. Оно отражает структуру запроса и содержит информацию, которая определяет объекты базы данных, упомянутые в запросе (таблицы, поля, константы). Внутреннее представление запроса используется и преобразуется на следующих стадиях обработки запроса.

На второй фазе запрос в своём внутреннем представлении подвергается логической оптимизации. При этом могут применяться различные преобразования, улучшающие начальное представление запроса.

Преобразования могут также использовать информацию об ограничениях целостности, существующих в БД. Такие преобразования являются семантическими, т. е. они основаны на семантике предметной области. В этом случае получаемое представление не является семантически эквивалентным начальному запросу. Но система гарантирует, что результат выполнения преобразованного запроса совпадает с результатом запроса в начальной форме при соблюдении ограничений целостности, существующих в базе данных. Например, если для таблицы emp определено такое ограничение целостности:

```
(CHECK (salary>15000 AND salary<100000)
```

то система к запросу из нашего примера могла бы добавить эти условия в часть `WHERE`, чтобы иметь возможность использовать индекс по полю `salary`.

После выполнения второй фазы обработки запроса его внутреннее представление является более эффективным, чем изначальный запрос, но остается непроцедурным. Это означает, что по этому представлению система сможет построить более эффективный план исполнения запроса.

Третий этап обработки запроса состоит в выборе альтернативных путей исполнения данного запроса в соответствии с внутренним представлением, полученным на второй фазе. Путь доступа, который существует всегда — это последовательное чтение всех строк таблицы (full scan). Наличие других путей доступа к данным зависит от наличия индексов, способов размещения данных, и формулировки самого запроса.

На этом же этапе каждому из планов исполнения сопоставляется стоимость выполнения запроса по данному плану. Оптимизатор при оценке стоимости исполнения пользуется доступной статистической информацией, либо информацией о механизмах реализации путей доступа. Из множества всех возможных планов исполнения запроса выбирается оптимальный с точки зрения некоторого критерия. Данный критерий выбирается заранее.

На четвертом этапе по внутреннему представлению выбранного оптимального плана исполнения запроса формируется его процедурное представление.

На пятом этапе обработки запроса происходит «выполнение» процедурного плана, а его результат помещается в контекстную область памяти.

Преобразования операций реляционной алгебры

Операции реляционной алгебры определяются на реляционных отношениях — неупорядоченных множествах кортежей. Формально, для получения результата любой из реляционных операций, необходимо проверить все кортежи в исходных отношениях. Однако, изменяя последовательность вычисления можно существенно уменьшить объём выполняемой работы.

Допустим, у нас есть два отношения A и B, мощность которых равна 10^4. Требуется вычислить эквисоединение этих двух отношений, после чего произвести выборку. Допустим также, что каждое из условий выборки затрагивает либо только атрибуты одного отношения, либо только атрибуты второго отношения (т. е. условия вида A.a+B.c=100 не допустимы), а также количество кортежей, удовлетворяющих условиям выборки составляет 10 кортежей в каждом отношении.

Формально, мы должны вычислить декартово произведение отношений, получив промежуточное отношение мощностью $(10^4)^2=10^8$(!) из которого должна быть произведена выборка.

Однако, если в начале из исходных отношений выбрать только те кортежи, которые удовлетворяют условиям выборки, то операндами для декартова произведения будут служить отношения с мощностью 10, а мощность результата будет 100 кортежей, что в 10^6 раз меньше.

Те же рассуждения можно применить и для операции объединения отношений.

Иными словами, меняя порядок вычисления результата можно существенно снизить трудоёмкость задачи.

Выражения реляционной алгебры считаются эквивалентными, если они описывают одно и то же отображение.

Основываясь на понятии эквивалентности реляционных выражений, можно выполнять оптимизацию вычисления реляционных операций.

Вот основные правила, которые определяют, как можно выполнять такого рода оптимизации:

1. Закон коммутативности для декартовых произведений:

$R_1 \times R_2 = R_2 \times R_1$

2. Закон коммутативности для соединений (P — предикат 1го порядка):

$(R_1$ JOIN R_2 condition: P) = (R_2 JOIN R_1 condition: P)

3. Закон ассоциативности для декартовых произведений:

$(R_1 \times R_2) \times R_3 = R_1 \times (R_2 \times R_3)$

4. Закон ассоциативности для соединений (P_1, P_2 — предикаты 1го порядка):

((R_1 JOIN R_2 condition: P_1) JOIN R_3 condition: P_2) = (R_1 JOIN (R_2 JOIN R_3 condition: P_2) condition: P_1)

5. Комбинация селекций (каскад селекций):

SELECT (SELECT R condition: P_2) condition: P_1 = SELECT R condition: (P_1 AND P_2)

6. Комбинация проекций (множество $\{A_m\}$ вложено в $\{B_n\}$):

PROJECT (PROJECT R (B_1, .. B_n)) (A_1, .. A_m) = PROJECT R (A_1, .. A_m)

7. Перестановка селекции и проекции:

SELECT (PROJECT R (A_1, .. A_m)) condition: P = PROJECT (SELECT R condition: P) (A_1, .. A_m)

8. Перестановка селекции с объединением:

SELECT (R_1 U R_2) condition: P = (SELECT R_1 condition: P) U (SELECT R_2 condition: P)

9. Перестановка селекции с декартовым произведением:
   - Если P = P_1 AND P_2, где P_1 содержит атрибуты, присутствующие только в R_1, а P_2 содержит атрибуты, присутствующие только в R_2:

SELECT ($R_1 \times R_2$) condition: P = (SELECT R_1 condition: P_1) × (SELECT R_2 condition: P_2)

- Если P содержит атрибуты, присутствующие только в R_1

```
SELECT (R1 × R2) condition: P = ( SELECT R1 condition: P ) × R2
```

- Если P содержит атрибуты, присутствующие только в R_2

```
SELECT (R1 × R2) condition: P = R1 × ( SELECT R2 condition: P )
```

- Если $P = P_1$ AND P_2, где P_1 содержит атрибуты, присутствующие только в R_1, а P_2 содержит атрибуты, присутствующие и в R_1, и в R_2

```
SELECT (R1 × R2) condition: P = SELECT ( ( SELECT R1 condition: P1 ) × R2) condition: P2
```

10. Перестановка селекции с разностью:

```
SELECT ( R1 – R2 ) condition: P = ( SELECT R1 condition: P ) – ( SELECT R2 condition: P )
```

11. Перестановка проекции с декартовым произведением (множества атрибутов $\{B_i\}$ и $\{C_j\}$ вложены в $\{A_n\}$, кроме того, атрибуты B_1, .. B_i представлены в отношении R_1, а атрибуты C_1, .. C_j – в R_2):

```
PROJECT (R1 × R2) (A1,A2,...,An) = ( PROJECT R1 (B1, .. Bi) ) × ( PROJECT R2 (C1, .. Cj) )
```

12. Перестановка селекции с пересечением:

```
SELECT (R1 INTERSECT R2) condition: P = ( SELECT R1 condition: P) INTERSECT (SELECT R2 condition: P).
```

Методы оптимизации

Существуют два альтернативных подхода к оптимизации запросов внутри СУБД.

Первый из них основывается только на информации о возможных путях доступа к данным и механизмах их реализации. Такой метод оптимизации, по сути, основан на синтаксисе и определённых правилах, как выбирать оптимальный путь доступа к данным. Отсюда и название этого метода оптимизации: rule-based optimization. Данный подход к оптимизации появился очень давно, и все современные СУБД имеют rule-based оптимизаторы.

Если же дополнительно к описанному оптимизатор учитывает накладные расходы на доступ к данным, то такой подход к оптимизации называется стоимостным (cost-based). Информация о накладных расходах основывается на статистических данных, которые собираются СУБД во время ее работы.

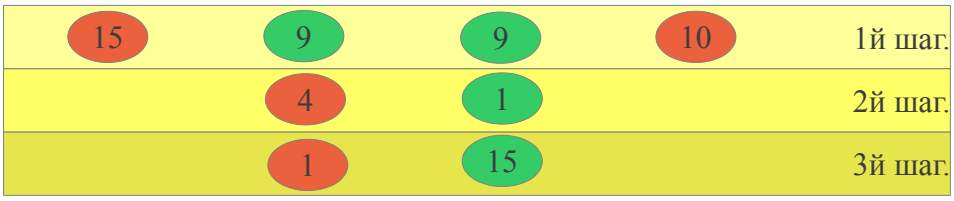
Rule-based оптимизация (RBO).

Каждому плану исполнения запроса сопоставляется определённая последовательность шагов-действий, которая должна быть выполнена для получения результата. Каждому типу таких шагов соответствует определённый весовой коэффициент, который обычно называется рангом операции. Последовательно рассматривая все шаги, на каждом из них rule-based оптимизатор выбирает те планы исполнения, весовой коэффициент которых наименьший.

Ниже указаны ранги путей доступа для СУБД Oracle8i.

Ранг	Путь доступа
1	Одна строка по ROWID (идентификатору строки)
2	Одна строка по кластерному соединению
3	Одна строка по хеш-кластеру с уникальным или первичным ключом
4	Одна строка по уникальному или первичному ключу
5	Кластерное соединение
6	Ключ хеш-кластера
7	Ключ индексного кластера
8	Составной индекс
9	Индекс по одиночному столбцу (по условию равенства)
10	Индексный поиск по закрытому интервалу
11	Индексный поиск по открытому интервалу
12	Сортировка-объединение
13	MAX и MIN по индексированному столбцу

14	ORDER BY по индексированному столбцу
15	Полный просмотр таблицы



Cost-based оптимизация (CBO).

При этом методе оптимизатор предварительно строит несколько потенциальных планов исполнения. Для выбора наиболее перспективных планов оптимизатор использует определённые эвристики. Эвристические правила позволяют отбросить неэффективные планы исполнения в самом начале и сузить пространство поиска оптимального плана. Эвристики, как правило, основаны на законах преобразования операций реляционной алгебры. В качестве примера можно привести такую эвристику: план является хорошим, если селекция производится на ранней стадии выполнения. Для каждого из построенных планов рассчитывается его стоимость.

Стоимость – это оценка ожидаемого времени выполнения запроса с использованием конкретного плана выполнения.

При расчёте стоимости оптимизатор может учитывать такие параметры, как количество необходимых ресурсов памяти, время операций дискового ввода-вывода, время процессора. Из множества возможных планов выполнения запроса оптимизатор в соответствии с критерием выбирает лучший план.

Критерием оптимизации может быть:

- Наилучшая общая производительность системы.
- Минимальное время реакции, т. е. время, необходимое для обработки и выдачи первой строки.
- Минимальные затраты времени на обработку всех строк, к которым обращается данная команда.

Стоимость плана выполнения запроса определяется на основании статистической информации о данных, к которым обращается команда: таблицах, индексах и т. п. Например, статистика о таблице может включать:

- общее число блоков данных, выделенных таблице;
- число пустых блоков данных;
- число записей в таблице;
- среднюю длину строки в таблице;
- среднее число строк на блок данных.

Возможны разные пути сбора статистики. СУБД могут собирать статистическую информацию постоянно, либо осуществляют ее сбор периодически (например, ночью при минимальной загрузке системы), либо же предоставляют администратору базы данных специальные утилиты для сбора статистики, которые необходимо запускать в ручном режиме.

Оптимизация клиентских приложений.

Цель этой категории оптимизаций — повышение эффективности взаимодействия с БД. К этой категории относятся работы по:

- настройке команд SQL;
- созданию индексов;
- выбору метода оптимизации SQL-запросов, если СУБД позволяет одновременно использовать RBO и CBO.

Настройка SQL–команд.

Тонкая настройка команд SQL является одним из самых сложных и в тоже время весьма эффективным способом повышения производительности. В идеале, такая настройка должна выполняться каждым разработчиком программного обеспечения.

Для её выполнения необходимо иметь точное представление о том, как именно и в каком порядке выполняются запросы в СУБД. Необходимо помнить, что различные СУБД могут существенно отличаться в этой части друг от друга, и решая оптимизационные задачи разработчики должны постоянно сверять свои действия с технической документацией к используемой СУБД.

Ниже приводятся некоторые рекомендации по правильному написанию запросов, в равной мере относящиеся ко многим СУБД. Основная цель этих рекомендаций — привести SQL запрос к виду, который будет удобен для оптимизатора СУБД.

1. Условия в запросе должны располагаться в порядке уменьшения селективности. Т. е. первым должно идти то условие, которому удовлетворяет наименьшее число строк.
2. Если СУБД использует RBO, то при запросе соединения нескольких таблиц, во фразе FROM таблицы должны указываться в списке FROM в порядке уменьшения количества записей в них, а в части WHERE первым должно стоять условие на родительскую таблицу.
3. Индексы допускают частичное использование своих ключей для левых колонок. Т. е. если индекс основан на десяти колонках, а условие в WHERE использует только одну колонку, которая была указана самой первой при создании индекса, то данный индекс будет использоваться.
4. Если таблица содержит индекс, основанный на нескольких колонках, условия на значения этих колонок во фразе WHERE следует указывать в том же порядке, что и при создании индекса.
5. Следует использовать UNION ALL вместо UNION, если в объединяемых отношениях отсутствуют одинаковые записи (или наличие одинаковых записей не критично). Дело в том, что UNION вычисляется путем сортировки, которая может занять много времени, а UNION ALL сортировки не требует.
6. СУБД иногда плохо оптимизируют операцию OR, в этом случае запрос можно разбить на два подзапроса, объединённых через UNION, а при возможности, через UNION ALL.
7. Условие "не равно" ('<>') подавляет использование индекса. Поэтому, если значения индексированного столбца распределены неравномерно, следует заменять его комбинацией условий … '<' … OR … '>' …

Например, список сотрудников всех подразделений (совокупно, допустим, 27% от общего числа), кроме сотрудников подразделения №97 будет выглядеть так:

```
SELECT *
  FROM emp
WHERE dpt_id < 97
UNION ALL
SELECT *
  FROM emp
WHERE dpt_id > 97;
```

8. Использование имени колонки в выражении не позволяют использовать индекс. Такие выражения необходимо переписывать таким образом, чтобы имя колонки указывалось отдельно. Например, условие salary/1.13>100000 должно быть записано как salary>100000*1.13.

При настройке команд SQL важно помнить, что, настраивая одну из них, можно оказать влияние (и не всегда позитивное) на другие команды. Поэтому во время настройки необходимо периодически осуществлять регрессионное тестирование, т.е. повторный запуск уже протестированных команд для оценки времени их выполнения.

Многие СУБД позволяют просмотреть план выполнения запроса средствами администрирования. Так можно убедиться в том, что система использует построенные индексы для выполнения запросов.

Индексирование данных

Индекс — это структура, которая определяет соответствие значения ключа записи (атрибута или группы атрибутов) и местоположения этой записи на диске.

Индекс хранится отдельно от таблицы либо в отдельном файле, как в MySQL, либо в отдельном индексном сегменте, как в Oracle.

Индексирование используется для ускорения доступа к записям по значению ключа, что достигается за счёт:

1. упорядочивания значений индексируемого атрибута. Это позволяет просматривать в среднем половину индекса при линейном поиске;
2. индекс занимает меньше памяти, чем сама таблица, поэтому СУБД тратит меньше ресурсов на чтение индекса, чем на чтение таблицы.

Индексы не влияют на размещение данных в таблицах. Поддержание и использование индекса происходит динамически и прозрачно для пользователя, т. е. после изменения данных в таблице — добавления или удаления записей, модификации индексируемых полей, — индекс приводится в соответствие с последней версией данных. Для каждой таблицы можно одновременно иметь несколько индексов, но поддержание индекса сопровождается определёнными накладными расходами, поэтому наличие избыточного числа индексов может существенно снизить производительность системы.

Обращение к записи таблицы через индекс выполняется в два этапа: во-первых, СУБД находит в индексе требуемое значение атрибута и определяет местоположение записи на диске, затем происходит обращение к диску на предмет доступа к данным.

Для таких ограничений целостности как: первичный ключ, уникальный ключ, ссылка на первичный ключ, СУБД создаёт индексы автоматически.

Организация индексов на основе B-дерева.

B-дерево — это сбалансированное сильно ветвящееся дерево, каждый узел которого, кроме корня, может содержать от n до 2n элементов-ключей, каждый узел может быть либо листом, либо иметь m+1 потомков, где m — число ключей, находящихся в узле.

Подробное описание B-дерева и алгоритмов работы с ним есть у Вирта в книге «Алгоритмы и структуры данных».

В реально используемых СУБД организация индексов отличается от классического B-дерева, т. к. учитывает наличие механизмов транзакций, в целях увеличения производительности допускается упреждающее расщепление узла и т. п. Такие деревья принято называть B^+-деревьями (B^*-).

Использование индексов

В системах, поддерживающих язык SQL, индекс создаётся командой CREATE INDEX:

```
CREATE INDEX <idx_name>
ON table_name ( column_name_list )
;
```

СУБД будет обращаться к составному индексу только тогда, когда в условиях выбора участвуют столбцы, представляющие собой начальную часть составного индекса. Например, если индекс был построен на колонках (A, B, C), то обращение к индексу будет происходить только в случаях, когда условие запроса поля ABC, AB либо A, причём порядок указания полей в условии имеет значение.

Селективность — критерий определяющийся процентом строк, имеющих одинаковое значение индексируемого столбца (столбцов): чем выше этот процент, тем меньше селективность.

Выбор столбцов подлежащих индексированию определяется следующими критериями:

- должны индексироваться столбцы, часто встречающиеся в условиях поиска;
- должны индексироваться столбцы, использующиеся для соединения таблиц;
- неразумно индексировать столбцы с низкой селективностью, за исключением случая, когда часто производится выборка по редко встречающимся значениям;
- неразумно индексировать часто обновляемые столбцы;
- неразумно индексировать столбцы, использующиеся как аргументы выражений или функций: как правило, такой индекс использоваться не будет.

Надо учитывать, что несколько столбцов таблицы с низкой селективностью, в комбинации могут обладать высокой селективностью, и могут использоваться для создания хорошего составного индекса.

Если в запросе используются только столбцы, участвующие в индексе, СУБД может вообще не обращаться к таблице.

Практические рекомендации.

При решении задач по оптимизации клиентских приложений рекомендуется использовать следующий (шутливый) алгоритм:

1. в обязательном порядке читать первую часть книги «Oracle. Оптимизация производительности.» авторства К. Миллсап;
2. если используемая СУБД — Oracle, читать остаток книги, если используется иная СУБД — искать информацию о способах профилирования исполняемых запросов в рамках используемой СУБД (например, «MySQL. Оптимизация производительности.» Шварц, Зайцев и др.)
3. основываясь на результатах профилирования запроса, выделить «узкое место» в работе запроса и определить пути решения проблемы.

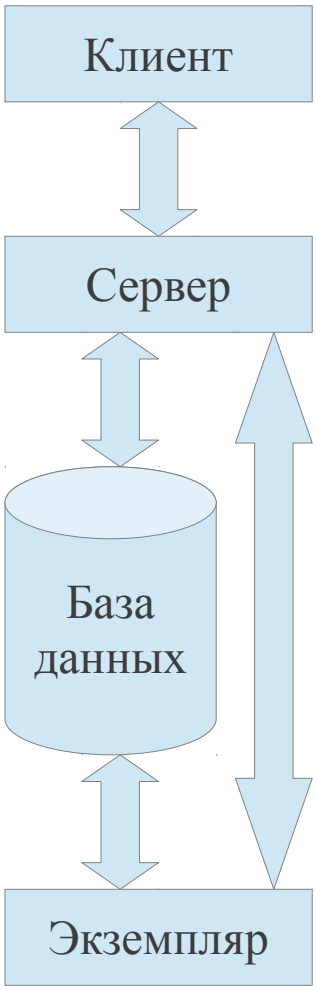
Архитектура СУБД Oracle.

Архитектура СУБД Oracle имеет три составляющие:

4. файлы,
5. процессы,
6. структуры памяти, используемые процессами.

Однако, рассматривать архитектуру СУБД Oracle принято с несколько иных позиций. При таком рассмотрении выделяется:

- собственно база данных, образованная, как правило, файлами нескольких типов, которые будут рассмотрены позднее;
- экземпляр Oracle, образованный фоновыми процессами и областью оперативной памяти, которая называется SGA;
- серверные процессы Oracle, обеспечивающие взаимодействие СУБД с прикладными программами.



Такое деление вызвано тем обстоятельством, что Oracle допускает работу с базой данных одновременно нескольких экземпляров. Такое возможно, например, при использовании дискового массива, доступ к которому имеют одновременно несколько серверов (технологии SAN, iSCSI и т. п.). На дисковый массив выносят базу данных, на каждом из серверов запускают экземпляры Oracle с дополнительной опцией Oracle Parallel Server, которая обеспечивает взаимодействие узлов получающегося кластера, после чего с базой данных одновременно работает несколько экземпляров. Однако, в подавляющем большинстве случаев, конкретная база данных всегда обслуживается единственным экземпляром.

Экземпляр Oracle.

Как уже было сказано, экземпляр Oracle (Oracle instance), образуется фоновыми процессами и SGA.

Структура System Global Area (SGA).

В зависимости от операционной системы, на которой работает Oracle, SGA может быть представлена либо разделяемой памятью (shared memory), например на операционных системах семейства UNIX, либо быть внутренней памятью процесса аллоцированной через функцию malloc языка C, как это сделано для операционной системы Windows. Однако вне зависимости от этого, внутренняя организация SGA на любой архитектуре выглядит одинаково. И так, SGA содержит следующие структуры памяти:

- Fixed area — область памяти в SGA, которая содержит значения различных параметров и указатели на остальные структуры SGA. Образ fixed area создаётся на момент установки программного обеспечения СУБД Oracle и отображается в память всякий раз при запуске экземпляра.
- Java pool — фиксированная часть памяти, которая выделяется виртуальной машине Java, работающей в составе СУБД.
- Large pool — используется серверным процессом для хранения User Global Area (UGA), в случае, когда серверный процесс работает в режиме shared server; при резервном копировании средствами утилиты rman для буферизации дискового ввода/вывода; для расположения буферов сообщений при использовании Oracle Parallel Server.

- Shared pool — область памяти в SGA, используемая для кеширования данных, связанных с обработкой SQL предложений (парсинг, оптимизация и т. п.). Внутри shared pool выделяется две части:
  - Shared SQL Cache, хранящий текст предложения SQL, как оно было получено от клиентского приложения, результат работы парсера для этого предложения, его план исполнения.
  - Data Dictionary Cache — область памяти, кеширующая результаты работы со словарём базы данных.
- Null pool — самая большая часть SGA, содержащая следующие структуры: буфер журнала повторного выполнения (redo log buffer) и буферный кеш (database buffer chache):
  - Буфер журнала повторного выполнения используется для оптимизации дискового ввода/вывода при работе с оперативным журналом повторного выполнения. Минимальный размер этого буфера составляет четыре максимальных размера блоков базы данных для данной архитектуры. Максимальный размер неограничен, но использование больших значений бессмысленно, т. к. большой буфер заполняться не будет. Содержимое этого буфера сбрасывается на диск по выполнению любого из условий:
    - один раз в три секунды;
    - при выполнении команды COMMIT;
    - если в буфере накапливается один мегабайт изменённых данных;
    - если буфер наполнился на одну треть от своего размера.
  - Буферный кеш хранит блоки базы данных, которые были прочитаны из базы при обработке запроса, размер этого буфера может составлять на современных системах десятки гигабайт. Для работы с данным буфером Oracle поддерживает два списка: список «грязных» блоков, т. е. блоков, данные в которых подверглись модификации, и список «чистых» блоков. Данные списки используются алгоритмами выбора жертвы, когда возникает необходимость освободить пространство в буферном кеше. Реально используемые алгоритмы постоянно меняются от версии к версии, но их основная идеология соответствует алгоритму LRU (Least Recently Used) — т. е. из буфера удаляются редко используемые блоки данных.

Фоновые процессы.

Фоновые процессы Oracle в зависимости от используемой операционной системы могут быть представлены либо реальными процессами операционной системы, как это сделано на UNIX-подобных системах, либо набором нитей исполнения, как это реализовано в Windows. Используя далее термин «процесс», будем подразумевать именно эти потоки исполнения.

Системный монитор (System Monitor, SMON)

Обеспечивает общую исправность экземпляра. SMON восстанавливает экземпляр при запуске после сбоя, координирует доступ и реализует восстановление экземпляра, когда к базе данных обращаются сразу несколько экземпляров. SMON также объединяет смежные области свободного пространства в файлах данных и освобождает пространство, занятое временными объектами, когда надобность в них отпадает (например пространство для сортировки строк).

Монитор процессов (Process Monitor, PMON)

Контролирует функционирование всех процессов Oracle и либо перезапускает их в случае необходимости, либо завершает работу экземпляра в случае серьёзного сбоя. Также PMON отвечает за очистку оставшихся занятыми ресурсов (таких как оперативная память) и за снятие всех блокировок, установленных сбойным процессом.

Процесс записи в базу данных (Database Writer, DBWR, DBWn)

Данный процесс предназначен для записи блоков данных из буферного кэша SGA в файлы данных, расположенные на диске. Один экземпляр Oracle может иметь до 20 процессов DBWR, обрабатывающих ввод/вывод в различные файлы данных, отсюда и обозначение DBWn. Запись блока на диск может происходит по двум причинам:

1. Когда необходимо прочитать блоки, запрошенные пользователем, а в кэше буферов нет свободного пространства. В этом случае на диск сбрасываются редко используемые блоки.
2. При выставлении контрольной точки, т. е. для обновления блоков файлов данных, с тем чтобы они «догнали» журнал повторного исполнения. Oracle записывает в журнал повторного исполнения информацию, необходимую для повторного выполнения транзакции после ее фиксации, а позже сохраняет сами блоки. Периодически Oracle выставляет контрольную точку для того, чтобы привести содержимое файла данных в соответствие с информацией, записанной в журнал для зафиксированных транзакций.

Процесс записи в журнал (Log Writer, LGWR)

Записывает информацию из буфера журнала повторного выполнения во все копии текущего журнального файла на диске. Всё время обработки транзакции, необходимая для ее повторного выполнения информация хранится в буфере журнала повторного выполнения. После выполнения команды COMMIT и завершения транзакции, процесс LGWR записывает эту информацию на диск для постоянного хранения.

Ввиду исключительной важности журналов повторного выполнения, для задачи восстановления базы после сбоя, обеспечению безопасности их содержимого уделяется особое внимание. Прежде всего, каждый из файлов журнала повторного выполнения должен иметь свои зеркальные копии на нескольких, физически разных, дисках. Процесс записи в журнал последовательно и циклически переписывает содержимое этих журнальных файлов. Таким образом, для журнальных файлов определяется три возможных состояния: активный (в который ведётся запись журнала прямо сейчас), неархивированный (полностью записанный, но еще не скопированный в архив) и свободный (данные которого уже скопированы в архив могут быть перезаписаны). Если в процессе таких переключений процесс LGWR не в состоянии переключиться на свободный журнальный файл, работа базы приостанавливается до момента, пока журнальный файл не будет помечен как свободный.

Архиватор (Archiver, ARCH)

Копирует заполненные файлы журнала повторного выполнения, находящиеся в состоянии «неархивированный» в архив. Архив журналов повторного выполнения может содержать до десяти зеркальных копий. По мере необходимости LGWR запускает дополнительные архиваторы в зависимости от нагрузки.

Контрольная точка (Checkpoint, CKPT)

Обновляет заголовки файлов данных после выставления контрольной точки.

Алгоритм двухфазной фиксации (two phase commit, 2PC).

В соответствии с алгоритмом 2PC, при получении команды COMMIT, все узлы, участвующие в распределённой транзакции, переходят в состояние PREPARE и подтверждают свой переход узлу–инициатору. Если хотя бы один узел не может перейти в состояние PREPARE — транзакция отменяется. Если все узлы успешно перешли в состояние PREPARE, узел–инициатор просит выполнить фиксацию изменений.

Данный механизм фиксации распределённых транзакций выполняется прозрачно для пользователей базы данных и не требует никаких дополнительных действий с их стороны.

Процесс восстановления (Recover, RECO)

Алгоритм двухфазной фиксации прекрасно работает при устойчивой работе сети, связывающей узлы распределенной транзакции. Однако, если в момент, когда все узлы перешли в состояние PREPARE связность сети будет нарушена, транзакция не сможет завершиться и перейдет в состояние IN DOUBT.

Процесс восстановления автоматически очищает неудавшиеся и отложенные распределенные транзакции.

Прочие фоновые процессы.

Существует еще целый ряд фоновых процессов, которые мы не рассмотрели. Некоторые из них возникают только при установленной опции Oracle Parallel Server. Некоторые выполняют сервисные функции, и не являются ключевыми в смысле функционирования архитектуры.

Отдельного упоминания заслуживает только фоновый процесс Listener, который принимает входящие соединения от клиентского программного обеспечения, и передаёт их на обслуживание в серверный процесс.

Серверный процесс Oracle.

Серверный процесс Oracle, обеспечивает взаимодействие СУБД с прикладной программой. При поступлении запроса к базе данных, именно серверный процесс выполняет работы по разбору предложения, поиску такого же предложения в shared pool, помещению всех необходимых данных в shared pool, если нужного предложения не было найдено. Именно он создаёт план исполнения и выполняет его. Серверный процесс ищет нужные данные в буферном кеше, и поднимает их с диска в кеш, если они еще не были закешированы. Серверный процесс выполняет сортировки, суммирования и прочее, прочее, прочее… Одним словом, всё, что связано с обслуживанием клиентов, выполняется серверным процессом.

Существует два режима работы серверных процессов: выделенный сервер (dedicated server) и разделяемый сервер (shared server, MTS).

В первом случае, для каждого входящего клиентского соединения фоновым процессом Listener порождается отдельный серверный процесс, который обслуживает только этого клиента.

Во втором случае, входящее клиентское соединение процессом Listener перенаправляется на специальный фоновый процесс «Диспетчер» (Dispatcher), т. е. все клиентские соединения попадают на этот процесс. Диспетчер принимает запросы клиентов и помещает их в очередь запросов. Разделяемый сервер, обслуживая эту очередь, извлекает очередной запрос, обрабатывает его и помещает ответ в очередь ответов. Диспетчер, в свою очередь, извлекает готовые результаты из очереди ответов и возвращает их нужному клиенту.

С точки зрения экземпляра между этими режимами работы серверного процесса есть существенная разница, состоящая в следующем.

С любой клиентской сессией связана определённая информация, описывающая контекст этой сессии, и расположенная в специальной структуре, которая называется UGA (User Global Area).

Любой серверный процесс имеет ассоциированную с ним область памяти, называемую Process Global Area (PGA). В PGA хранится информация процесса: располагаются области сортировки, области хешей и другие структуры.

При использовании выделенного сервера весь контекст сессии клиента, т. е. UGA, хранится в PGA процесса, обслуживающего данного клиента.

При использовании разделяемого сервера весь контекст сессии клиента, хранится в большом пуле в SGA.

Файлы базы данных.

Oracle имеет шесть типов файлов. Один из этих типов — **файлы параметров** — связан с экземпляром. На основании этих файлов, во время старта экземпляра, СУБД определяет свои настройки, например размер SGA и ее частей, местонахождение управляющих файлов базы данных и т. п.

Остальные пять типов файлов образуют, собственно, базу данных Oracle:

Файлы данных.

Собственно файлы с данными, в которых располагаются таблицы, индексы и все объекты, хранящиеся в базе данных.

Управляющие файлы.

Содержат ключевую информацию о физической структуре базы данных: местонахождение файлов данных и файлов журнала повторного выполнения. Содержат другую необходимую информацию о состоянии базы данных.

Файлы журнала повторного выполнения.

Уже рассматривались в разделе, описывающем фоновый процесс LGWR.

Временные файлы.

Используются для размещения временных объектов, например при сортировке больших объемов данных.

Файлы паролей.

Используются для аутентификации пользователей. Мы их рассматривать не будем.

Первые три типа файлов, чрезвычайно важны, т. к. именно они содержат информацию о физической структуре СУБД и в них хранятся накопленные данные.

Если потеряны управляющие файлы, потери данных нет, но восстановление работы базы становится нетривиальной технической задачей.

При утере файлов журнала повторного выполнения, будут потеряны только те данные, которые находились в самом старом из утерянных журналов, а так же все последующие модификации данных.

В случае, если будут утеряны файлы данных, информацию можно восстановить только при наличии резервных копий.

Файлы данных.

Как следует из названия, данные в Oracle хранятся в файлах данных. Это верно с точки зрения физической структуры базы данных, но кроме это уместно упомянуть о логической структуре хранения данных.

С логической точки зрения, все объекты в базе данных хранятся в **табличных пространствах**. Табличное пространство является логической абстракцией, позволяющей

скрыть от пользователей базы данных технические подробности физической организации хранения данных. Табличное пространство определяется на одном или нескольких файлах данных, на «сырых» дисках (т. е. на неотформатированных разделах жестких дисков) и т. п.

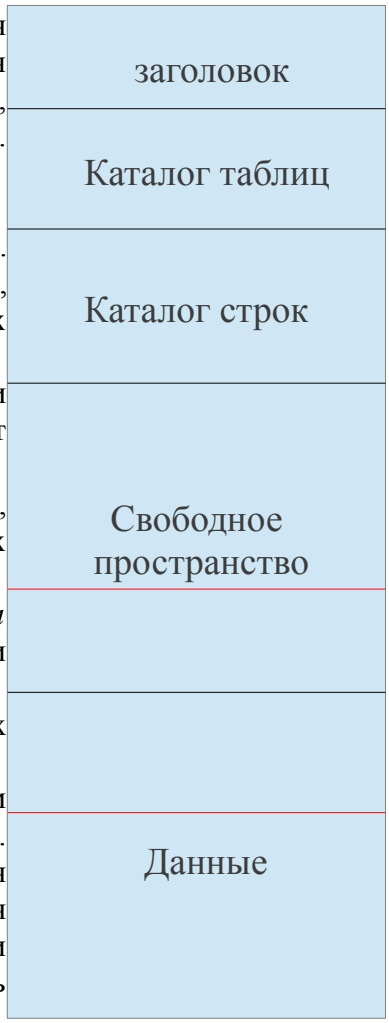
С каждым из объектов, хранящемся в СУБД (таблицы, индексы, материализованные представления и т. д.), связана область на диске, выделенная под хранение этого объекта. Такую область в терминологии Oracle называют **сегментом**. В случае, если хранимый объект является секционированным (partitioned), под каждую секцию создаётся свой сегмент.

Табличное пространство является контейнером сегментов. Любой сегмент всегда принадлежит только одному табличному пространству, табличное пространство может содержать произвольное число сегментов.

Экстент — это непрерывный участок дискового пространства в файле. Если в сегменте не хватает свободного пространства для размещения данных, сегмент расширяется путём аллокации очередного экстента и присоединения его в данному сегменту. Каждый сегмент состоит, как минимум, из одного экстента. Однако, некоторые объекты требуют наличия минимум двух экстентов, например назвать сегменты отката. Каждый следующий экстент не обязательно должен выделяться рядом с предыдущим; он может находиться достаточно далеко от первого. Более того, если табличное пространство построено на основе нескольких файлов данных, то экстенты одного сегмента могут располагаться во всех этих файлах данных. Но, подчеркнём это еще раз, в пределах экстента пространство файла всегда непрерывно.

Блок базы данных — наименьшая единица выделения пространства в Oracle. Дисковый ввод/вывод выполняется блоками, данные хранимых и временных объектов — таблиц, индексов, результаты сортировок — хранятся именно блоками. Размер блока данных в Oracle бывает: 2, 4, 8, 16 и 32 Кбайта.

Формат блока представлен на рисунке.



Заголовок блока содержит информацию о типе блока (т. е. какому типу объекта принадлежит блок: таблице, индексу и т. д.), информацию о текущих и прежних транзакциях, затронувших блок, а также местонахождение блока на диске.

Каталог таблиц содержит информацию о таблицах, строки которых хранятся в этом блоке (в случае кластера блоке могут храниться строки из таблиц).

Каталог строк является массивом указателей на строки, хранящиеся в области данных блока, и содержит описание этих строк.

Эти три части блока, совокупно, называются *служебным пространством блока*. Оно недоступно для данных и используется Oracle для управления блоком.

Для каждого сегмента, Oracle поддерживает списки занятых и свободных блоков.

Если в блоке остаётся меньше свободного пространства, чем указано в параметре PCTFREE, блок помечается как занятый. Если из занятого блока удаляются данные, блок будет считаться занятым, пока размер данных не опустится ниже значения параметра PCTUSED (Красные линии на рисунке). При необходимости параметры PCTFREE и PCTUSED можно указать явно при создании хранимых объектов.

PCTFREE и PCTUSER указываются в процентах, их сумма должна всегда быть меньше 100%. Зачем существует PCTFREE и почему заполнять блок данных „под завязку” не самая

хорошая идея?

Допустим, в таблице есть колонка VARCHAR(1000), и конкретной ячейке из данной колонки лежит строка '1'. Данная строка занимает один байт, и, формально, мы имеем право дописать в эту строку еще 999 байт. Что будет при этом, если места в блоке данных не хватит?

В этом случае на месте строки оставляется специальный маркер с указанием нового месторасположения строки. Это приводит к дополнительным операциям ввода-вывода, что снижает производительность. Данное явление носит название миграции строк.

Его ни в коем случае нельзя путать с нормальным явлением под названием row chaining: в случае, если размер строки превосходит размер блока данных, Oracle разделит данные на несколько частей, и запишет данные в несколько блоков данных.

Временные файлы.

Временные файлы (временные табличные пространства) используются для хранения временных объектов, например для сортировки результатов запроса. Создать в них какие-либо объекты не возможно.

Управляющий файл.

Управляющий файл содержит информацию обо всех файлах, необходимых серверу Oracle. Файл параметров инициализации (init.ora) указывает экземпляру, где расположен управляющий файл. В управляющем же файле описано местонахождение файлов данных, файлов журнала повторного выполнения, хранятся время обработки контрольной точки, имя базы данных, дата и время создания базы данных, хронология архивирования журнала повторного выполнения, записи, вносимые программой резервного копирования RMAN и т.д.

Обработка транзакций в Oracle.

Рассмотрим, как работает Oracle при обработке DML команд..

Прежде всего, транзакция в Oracle начинается с любой из команд: INSERT, UPDATE, DELETE, SELECT FOR UPDATE, MERGE, LOCK TABLE и заканчивается либо командами DDL, либо командами ROLLBACK/COMMIT.

Допустим, мы выполняем операцию вставки данных. С диска в SGA будет поднят блок базы данных, в который будет выполнена вставка (записаны новые данные). Такой блок после внесения изменений становится „грязным”. Перед записью новых данных в блок, СУБД создаст записи в redo-журнале и в undo-сегменте. Первой будет достаточно, чтобы воспроизвести изменения повторно, а второй — для отмены изменений в случае отката транзакции. Вся информация при этом кешируется (и грязный блок и redo-, undo-записи).

Если прямо после это случится крах системы, то всё хорошо — транзакция не закончилась, следовательно, при восстановлении мы должны ее откатить, но откатывать еще нечего, т. к. данные не были записаны на диск.

Если буферный кеш переполняется, и наш блок данных должен быть записан на диск: процесс DBWR, который должен записать грязные данные на диск даёт указание процессу LGWR записать на диск все redo-записи, связанные с нашим блоком данных. В том числе будут записаны redo-записи для undo-записей. Если в этот момент случится крах системы то на основе redo-информации можно будет восстановить и грязный блок и undo-записи, с ним связанные.

Что будет при **ROLLBACK**? СУБД на основе undo-записей восстановит исходные значения блоков данных, и со временем они будут записаны в файлы данных, если в этом будет необходимость. Redo-информация при этом вообще не затрагивается!

А что будет при **COMMIT**? Тогда кеш redo-журнала будет записан на диск, а

транзакция помечена как завершённая. Если при этом произойдёт крах, мы сможем восстановить транзакцию полностью. Если есть параллельные транзакции использующие старые данные, они будут брать их из undo-информации. По завершению всех транзакций, которые используют старые данные, undo-информация очищается. Новые данные из буферного кеша будут записаны на диск „по случаю.”

Современные нереляционные СУБД.

Существует много причин, по которым использование реляционной СУБД в конкретном проекте может быть нежелательным: недостаточная производительность, недостаточная масштабируемость, желание избежать процедуры отображения объектов на реляционную модель (*Object-Relational Mapping, ORM*) и т. п.

Условно, современные нереляционные СУБД можно разделить на несколько категорий:

Объектно-реляционные СУБД —
: позволяют хранить данные как в виде отношений, так и в виде объектов (без использования ORM). Например, Oracle или PostgreSQL. По сути, это попытка решить проблемы чисто реляционных баз данных путём расширения их модели данных.

Объектно-ориентированные СУБД —
: позволяют хранить объекты в «чистом виде». Для работы с такими СУБД может использоваться либо язык OQL (аналог языка SQL), либо специальные функции, предоставляемые драйвером конкретной СУБД. В примером таких СУБД являются: Cache, db4o и ObjectStore.

Колоночно-ориентированные —
: данные хранятся в столбцах. Это удобно для архивов информации и каталогов, в которых большая часть вычислений происходит над подобными выборками данных. Представителями этой модели являются BigTable и Cassandra.

Ассоциативные —
: все данные хранятся в виде пар ключ-значение. Широко известным представителем данной категории является MemcacheD.

Документно-ориентированные —
: каждая запись хранится как отдельный документ, имеющий собственный набор полей, который может отличаться от документа к документу. Популярные реализации такой модели — CouchDB и MongoDB.

Графовые —
: Используют вершины и ребра графа для представления информации. Очевидно, что такая модель гораздо производительнее работает с данными представленными в виде графов, например социальных графов. Наиболее известные системы хранилища данных такого типа — *neo4j, AllegroGraph, ActiveRDF.*

Каждая из этих категорий заслуживает отдельного большого описания, но мы рассмотрим только первые вторую и третью категории.

Объектно-ориентированные СУБД

Появлению объектно-ориентированных СУБД способствовали несколько факторов, которые можно условно разделить на проблемы связанные с объектно-ориентированными языками программирования и на недостатки существующих **не**объектно-ориентированных СУБД.

Причины появления ООСУБД.

Постоянство объектов.

В объектно-ориентированной системе объекты создаются во время работы системы. В рамках этой системы существует определённый жизненный цикл этих объектов. Однако, при прерывании работы объектно-ориентированной системы, по какой бы то ни было причине,

все объекты прекращают свое существование. Это называется отсутствием постоянства. Наличие базы данных, сохраняющей объекты, появляется постоянство, при котором существование объекта не связано с функционированием системы.

Ограничения со стороны оперативной памяти

Современные информационные системы оперируют данными, объем которых заведомо превышает объем наличной оперативной памяти. Наличие базы данных позволяет существенно снизить размер данных, хранящихся в оперативной памяти.

Совместный доступ к объектам

При использовании совместно используемой базы данных, каждый объект может разделяться между пользователями с естественными ограничениями, например, механизма предотвращения конфликтов при одновременном доступе к объекту. При отсутствии базы данных, реализовать такое поведение возможно, но трудозатраты на это будут превышать все разумные пределы.

Невозможность адекватного представления объектов в не объектно-ориентированных СУБД

Не объектно-ориентированная СУБД не позволяет хранить поведенческую модель объектов. Сложности, иногда очень существенные, при отображении объектов на реляционную модель.

Модель данных ООСУБД.

Модель данных в объектно-ориентированных СУБД подразделяется на структурную и поведенческую части. Структурная часть связанна с данными и их доменами и подразделяется на уровень данных и уровень схемы. Поведенческая часть определяет функциональную составляющую: функции, методы и исключения.

Уровень данных описывает данные как таковые: непосредственно данные, соотношения между ними, наличие уникального идентификатора у объектов и т. п.

На уровне схемы описываются: типы и классы. В отличие от традиционного ООП в ООБД принято различать типы и классы. Впервые такое разделение было предложено Беери, который отмечал, что во всех системах, использующих только одно понятие (либо тип, либо класс), это понятие неизбежно перегружено: тип предполагает наличие некоторого множества значений, определяемого структурой данных этого типа; класс также предполагает наличие множества объектов, но это множество определяется пользователем.

Поведенческая часть включает в себя понятия метакласса, классового атрибута, наследования, методов и функций.

ООСУБД на примере db4o

Наиболее заметными представителями современных объектно-ориентированных СУБД на текущий момент являются СУБД Caché и db4o. Первая из них является коммерческим продуктом компании InterSystems и имеет программные интерфейсы для многих объектно-ориентированных языков программирования, таких как: C++, Java, и т. п. Db4o является open source продуктом, который разрабатывается компанией Versant и имеет интерфейсы для Java и .NET.

В качестве примера рассмотрим работу с db4o из Java программы:

```
import com.db4o.*;

public class Db4oExample {

    public static void main ( String args[] ) {
```

```
        ObjectContainer db = Db4oEmbedded.openFile("some_db4o_db.yap");

        SomeClass object_1 = new SomeClass (1);
        SomeClass object_2 = new SomeClass (2);

        AnotherClass object_3 = new AnotherClass ();

        db.store(object_1);
        db.store(object_2);
        db.store(object_3);

        db.commit();

        //method 1
        SomeClass qbe_proto = new SomeClass ();
        ObjectSet result = db.queryByExample(qbe_proto);

        //method 2
        ObjectSet result = db.queryByExample(Pilot.class);

        SomeClass found_object = result.next();
        found_object.setField ( new_value );
        db.store ( found_object );

        found_object = result.next();
        db.delete ( found_object );

        //method 3
        List<SomeClass> objects = database.query<SomeClass> (
            new Predicate<SomeClass>() {
                public boolean match(SomeClass some_object) {
                    return some_object.getField1() > 0 &&
                           some_object.getField2().equals("string");
                }
            }
        );

        db.close();
    }
}
```

Приведённый выше код, откроет файл базы данных, а если его нет — создаст. Далее заводятся три объекта, которые будут сохранены в базе данных, после чего изменения фиксируются в базе данных. Db4o допускает несколько способов выборки данных из базы:

query by example (QBE),

native queries (NQ),

simple object database access (SODA)

Первый способ демонстрируется на примере участков кода, под комментариями «method1» и «method2». При таком способе доступа, выбираются данные «похожие» на эталонный объект, который по сути является значением фильтра выборки.

Способ доступа NQ показан на примере участка кода под комментарием method3

Обновление объекта в базе данных сводится к нахождению этого объекта и прочтению его в программу, модификации объекта и последующему сохранению этого объекта в базе данных.

С программной точки зрения удаление объекта из базы данных аналогично обновлению: нахождение объекта и собственно удаление.

Cassandra, как пример колончно-ориентированных баз данных.

Cassandra — это распределенная колоночно-ориентированная система управления базами данных, которая развивается в рамках проекта Apache. Данная СУБД проверена в работе такими гигантами IT-индустрии как Facebook, Twitter, Digg и в др., основными плюсами данной системы принято считать:

- высокая степень горизонтальной масштабируемости;
- высокая отказоустойчивость, т. к. система подразумевает развёртывание на кластерах, и каждая запись дублируется на нескольких узлах в кластере;
- децентрализована, т. е. все узлы в кластере одинаковы;
- производительность операций чтения/записи увеличиваются линейно с добавлением узлов в кластер;
- нет ограничений на объем базы, поиск данных всегда O(1);
- гибкая модель данных.

Прежде чем обсуждать остальные вопросы, остановимся на первом пункте. Как известно, масштабируемость информационных систем принято подразделять на горизонтальную и вертикальную. Вертикальная масштабируемость означает, что в случае нехватки ресурсов в системе, система должна заменяться на более мощную[30]. Горизонтальная масштабируемость означает, что в случае нехватки ресурсов в системе, система должна расширяться за счет установки новых мощностей дополнительно к уже имеющимся.

По своей архитектуре СУБД Cassandra изначально разрабатывалась в качестве кластерной системы, которая будет функционировать поверх кластеров с очень большим числом узлов. Таким образом, все плюсы, кроме последнего, являются следствием продуманной и аккуратно реализованной архитектуры.

Модель данных СУБД Cassandra.

Рассматривая модель данных аналогичных Cassandra надо „забыть” терминологию, используемую в реляционной модели данных и реляционных СУБД. Попытки построить аналогии между объектами этих моделей данных, на основе одинаковых названий терминов неизбежно приведут к путанице и существенно усложнят понимание. Напротив, все описываемые объекты данной модели рекомендуется воспринимать с позиций ассоциативных массивов.

Базовым элементом в структуре данных СУБД Cassandra является колонка.

Колонка (Столбец, Column) —

это триплет следующих полей {name: byte[]; value: byte[]; time: byte[]}

Последнее поле используется самой СУБД при решении вопросов синхронизации данных, в приложениях почти никогда не используется, хотя приложения при внесении и модификации данных должны его явно указывать, как правило, его значением является значение времени в UNIX формате (количество секунд с 01.01.1970). Поэтому, при обсуждении структур данных в Cassandra, данное поле не упоминается и не обсуждается. Мы будем следовать этой практике. Т. е. с логической точки зрения колонка — это пара {name: b_string; value: b_string}, где name — идентификатор колонки, а value — её значение. Следующее упрощение записи приводит нас к виду {имя_колонки: значение_колонки}.

Cassandra игнорирует пользовательские типы данных (точнее она про них вообще не знает). Все данные, хранящиеся в базе, Cassandra воспринимает как массивы байтов, неограниченные по длине. Это означает, что в качестве имени и значения колонки могут выступать абсолютно любые данные. Например, в качестве имени может быть указана фотография участника научной конференции, а значением будет текст его доклада.

Суперколонка (Суперстолбец, SuperColumn) —

это колонка, которая в качестве значения принимает набор колонок.

Основное отличие колонки от суперколонки состоит в том, что колонка принимает атомарное значение (в том же смысле, что и в реляционных СУБД), а супер колонка позволяет хранить множество других колонок, т. е. ее значение составное.

Вторым отличием является отсутствие в суперколонке метки времени.

30 Как издевательски комментируют этот вариант масштабируемости воинствующие противники реляционных СУБД: „просто купи шкаф побольше!”

Строка — это уникальный идентификатор и соответствующий ему набор колонок. Данный элемент модели данных может существовать только в рамках семейства столбцов, которое описано ниже.

Семейство столбцов (Column Family) — это коллекция строк.

При описании семейства столбцов в storage-conf.xml указывается параметр CompareWith, указывающий правила сортировки колонок внутри строки. По-умолчанию, Cassandra предлагает следующие варианты типов сортировок: BytesType, UTF8Type, LexicalUUIDType, TimeUUIDType, AsciiType, и LongType. Для сортировки суперколонок можно указывать дополнительный параметр CompareSubcolumnsWith.

При необходимости, можно создать свой собственный компаратор, реализующий интерфейс org.apache.cassandra.db.marshal.IType.

Ниже показана структура данных, описывающая многопользовательскую адресную книгу.

```
AddressBook = {
    // это ключ строки внутри семейства суперстолбцов
    // ключом служит пользователя-владельца данной адресной книги
    owner1: {
        // строка содержит произвольное число суперстолбцов
        // ключами суперстолбцов в строке являются имена контактов
        // значением суперстолбцов является адрес данного контакта.
        contact1: {contry: "", region: "", city: "", street: ""},
        contact2: {contry: "", region: "", city: "", street: "", phone: ""},
        contact3: {contry: "", region: "", city: "", street: ""},
    }, // конец строки (адресной книги пользователя owner1)
    owner2: {
        …
    },
}
```

Обратите внимание, что для второго контакта определён номер телефона, который отсутствует у остальных контактов.

Пространство ключей (Keyspace) — это контейнер для семейств колонок. Все семейства столбцов находятся в том или ином пространстве ключей, если проводить грубые аналогии, то пространство ключей соответствует понятию схемы в реляционных базах данных.

Оглавление.

Оглавление

Литература.

1. К. Дейт «Введение в системы баз данных» // 2-е изд. – М.: Наука, 1980
2. К. Дейт «Введение в системы баз данных» // 7-е изд. – М.: Вильямс, 2001. – 1072 с.
3. К. Дейт «Введение в системы баз данных» // 8-е изд. – М.: Вильямс, 2005. – 1328 с.
4. Стандарт ISO/IEC 9075:1992 . «Database Language SQL". (Так же известен как ANSI X3.135-1992.)
5. Стандарт ISO/IEC 9075-2:1999 . «Database Language SQL".
6. С. Урман «ORACLE 9i. Программирование на языке PL/SQL.» // – М: Лори, 2004. – 535 с.
7. С. Урман «ORACLE 8. Программирование на языке PL/SQL.» // – М: Лори, 1999. – 607 с.
8. С. Урман, Р. Хардман, М. МакЛафлин «ORACLE DATABASE 10g. Программирование на языке PL/SQL» // – Спб: Питер, 2004. – 535 с.
9. М. Грубер «Понимание SQL» // 1993. – 291с.
10. MySQL 5.5 Reference Manual: http://dev.mysql.com/doc/refman/5.5/en
11. Oracle Documantation: http://www.oracle.com/technetwork/indexes/documentation/index.html
12. Документация по MySQL (интернет ресурс посвященный MySQL): http://mysql.ru/docs
13. MSDN Library: http://msdn.microsoft.com
14. Oracle SQL & PL/SQL: http://sql-plsql.blogspot.com
15. Oracle Internals: http://www.juliandyke.com/Presentations/Presentations.html
16. Е.Ф. Кодд «Реляционная модель для больших совместно используемых банков данных» // СУБД. 1995. -No.1. -С. 145-169. (перевод М.Р. Когаловского на CITcitforum)
17. Oracle® Database Concepts 11g Release 1 (11.1) Part Number B28318-05
18. Oracle® Database Administrator's Guide 11g Release 1 (11.1) Part Number B28310-04
19. Д. Кренке «Теория и практика построения баз данных» // – Спб: Питер, 2005. – 864 с.
20. Клайн К. при участии Клайна Д. и Хаита Бр. «SQL. Справочник.» // 2-е издание, – М.: КУДИЦ-ОБРАЗ, 2006. – 832 с.
21. П. Роб, К. Кренке «Системы баз данных: проектирование, реализация и управление» // – Спб: БХВ-Петербург, 2004. – 1040 с.
22. М. П. Малыхина «Базы данных. Основы, проектирование, использование» // – Спб: БХВ-Петербург, 2006. – 528 с.
23. А. Д. Хомоненко, В. М. Цыганков, М. Г. Мальцев «Базы данных. Учебник для вузов» // – Спб: Корона-принт, 2004. – 736 с.
24. Д. Энсор, Й. Стивенсон «Oracle. Проектирование баз данных» // – Спб: BHV, 1999. – 560с.
25. Т. Коннолли, К. Бегг «Базы данных. Проектирование, реализация и сопровождение. Теория и практика» // – М: Вильямс, 2003. – 1436 с.
26. Д. Энсор, Й. Стивенсон «Oracle. Проектирование баз данных» // – Спб: BHV, 1999. – 560с.
27. К. Милсап, Д. Хольт «Oracle. Оптимизация производительности» // – М: Символ-Плюс, 2006. – 464 с.
28. Р. Нимик «Oracle9i. Оптимизация производительности. Советы и методы» // – М: Лори, 2006. – 726 с.
29. Б.Шварс и др. «MySQL. Оптимизация производительности» // – М: Символ-Плюс, 2010. – 232 с.
30. Д. Мюллер, И. Чоудри «Microsoft Windows 2000. Настройка и оптимизация производительности» // – М: Эком, 2001. – 512 с.
31. Т. Кайт. «Oracle для профессионалов. Архитектура, методики программирования и

основные особенности версий 9i и 10g.» // – М: Вильямс, 2007. – 848 с.
32. Р.Д. Яргер, Д. Риз, Т. Кинг «MySQL и mSQL. Базы данных для небольших предприятий и Интернета» // –М: Символ-Плюс, 2000. – 560 с.
33. E. Codd «"Is Your DBMS Really Relational?" and "Does Your DBMS Run By the Rules?"» // ComputerWorld, October 14 and October 21.
34. E. Codd «Relational Completeness of Data Base Sublanguages» // Database Systems, Courant Computer. Science Symposium Series 6. – Prentice-Hall, Englewood Cliffs, NJ, 1972
35. R. Elmasri, & S. Navathe, «Fundamentals of Database Systems» // 2nd ed., Redwood City, CA: The Benjamin/Cummings Publishing Co., 1994, pp. 283 – 285.
36. D. McLead «High Level Definition of Abstract Domains in a Relational Data Base System» // Comput. Lang. 2, Pergamon Press, 1977, pp. 61-73
37. Ш. Атре «Структурный подход к организации баз данных» // – М.: Финансы и статистика. 1983. – 313 с.
38. В.В. Корнеев, А.Ф. Гареев, С.В. Васютин, В.В. Райх «Базы данных. Интеллектуальная обработка информации» // 2-е изд. – М.: Нолидж, 2001. – 496 с.
39. М. Нагао, Т. Катаяма, С. Уэмура «Структуры и базы данных» // – М.: Мир, 1986. 197 с.
40. M. Stonebraker, L. Rowe, B. Lindsay, J. Gray, M. Carey, M. Brodie, Ph. Bernstein, D. Beech. “Third-Generation Data Base System Manifesto”. Proc. IFIP WG 2.6 Conf. on Object-Oriented Databases, July 1990, ACM SIGMOD Record 19, No. 3 (September 1990). (Имеется русский перевод: Стоунбрейкер М. и др. “Системы баз данных третьего поколения: Манифест”, СУБД, No. 2, 1996,
41. C.Beeri – A formal approach to object-oriented databases. – Data & Knowledge Engineering, vol 5 (1990), pp. 353-382
42. Тарасова И. А. Диссертация «Метод проектирования логической структуры реляционной БД без нормализации таблиц».
43. William H. Inmon. «Building the Data Warehouse, Fourth Edition.» Wiley Publishing Inc. // 2005. – 576 с.
44. Хоббс Л., Хилсон С., Лоуенд Ш. «Oracle 9iR2: разработка и эксплуатация хранилищ баз данных. Практическое пособие» // М.: КУДИЦ-ОБРАЗ, 2004. – 592 с.
45. Paulraj Ponniah. «Data Warehousing Fundamentals: A Comprehensive Guide for IT Professionals.» Wiley-Interscience Publication. // 2001. – 544 с.
46. Eben Hewitt. «Cassandra: The Definitive Guide.» O’Reilly. // 2011. – 330 с.