

Введение в Oracle SQL

Материалы учебного курса

Владимир Викторович Пржиялковский

open-oracle.ru

prz@yandex.ru

Март 2019

Москва

"... и ясен будет язык гугнивых".
Исаия 35:5,6. Цит. по: Митрополит Иларион.
Слово о Законе и Благодати

Оглавление

Введение в Oracle SQL.....	1
1. Происхождение и объем диалекта SQL фирмы Oracle.....	3
2. Инструменты для общения с базой данных.....	17
3. Данные для дальнейших примеров.....	19
4. Создание, удаление и изменение структуры таблиц.....	25
5. Общие элементы запросов и предложений DML: выражения.....	43
6. Выборка данных.....	61
7. Обновление данных в таблицах.....	146
8. Объявленные ограничения целостности.....	160
9. Представления данных, или же виртуальные таблицы (views).....	170
10. Объектные типы данных в Oracle.....	177
11. Вспомогательные виды хранимых объектов.....	190
12. Некоторые общие свойства объектов хранения разных видов.....	204
13. Некоторые замечания по оптимизации выполнения предложений SQL.....	210
14. Транзакции и блокировки.....	214
15. Таблицы словаря-справочника.....	225
16. Встроенный SQL.....	227
Некоторые примеры составления запросов.....	228
17. Запрос первых N записей.....	228
18. Декартово произведение.....	231
19. Ловушка условия с отрицанием NOT	233
20. Ловушка в NOT IN (S).....	234
21. Типичная ошибка в составлении полуоткрытых соединений.....	236
Страница для заметок.....	238

1. Происхождение и объем диалекта SQL фирмы Oracle

"Ну вот и все предисловие. Я совершенно согласен, что оно лишнее, но так как оно уже написано, то пусть и останется."

Ф. М. Достоевский. Братья Карамазовы.
Предисловие автора.

Для успешного программирования в Oracle на SQL недостаточно знать сугубо формальное описание языка. Необходимо владеть более широким набором имеющихся отношение к делу знаний. С этой целью ниже рассматривается цепочка понятий, подводящая к «диалекту SQL, предлагаемому фирмой Oracle». Она устроена следующим образом: база данных → модель данных, СУБД; реляционная модель; язык запросов к данным и изменения данных → SQL → диалект SQL в Oracle.

1.1. База данных и модель данных

1.1.1. База данных

В буквальном переводе на русский язык «база данных» (БД) означает специально подготовленную на компьютере «основу» (data) для работы потребителей с «данными» (data). Непосредственным потребителем является, конечно, программа.

Общепринятого понятия базы данных, несмотря на широкое распространение самого явления, не существует. Вот некоторые примеры разнохарактерных определений.

- (1) «Обычно большое собрание данных, организованных для особо быстрого и удобного способа поиска и извлечения (например из ЭВМ)» (*Merriam-Webster's Collegiate Dictionary*, www.merriam-webster.com/dictionary/database, датировано 1962 годом).
- (2) «Собрание структуризованных данных в ЭВМ, поддерживаемое СУБД, которая обеспечивает различным приложениям различный вид данных» (*F. Pascal, Understanding Relational Databases with examples in SQL-92, New York, NY: John Wiley & Sons, 1993*).
- (3) «База данных — набор аксиом. Результат на запрос к базе есть теорема. Процесс вывода теоремы из аксиом есть доказательство. Доказательство осуществляется манипулированием символов по условленным математическим правилам. Доказательство [то есть, результат запроса к базе] настолько же здраво и логично (consistent), насколько здравы и логичны правила» (*H. Darwen. The Duplicity of Duplicate Rows. Relational Database Writings 1989 — 1991, Reading, MA: Addison-Wesley, 1992*).

Возможное обобщение этих и других определений:

- (0) «Совокупность всех данных некоторой прикладной области» [для использования в программных системах] (*Филиппов В. И. Общее описание системы КОМПАС. // Автоматизация программирования. Москва: ВЦ АН СССР, 1989*).

Существенные элементы в определениях БД

- **Модель.** Всякая БД, независимо от того, создал это ее разработчик или нет, воплощает собой некоторую «модель данных» «предметной области»: понятийную (говоря по-иному, «концептуальную», «бизнес-модель»), логическую (данных) и физическую (организации данных)¹.
- **Собственно БД.** Организованные для долговременного хранения, обычно на внешнем носителе, данные общего пользования.

¹ Иногда тройку «концептуальная», логическая и физическая модель выстраивают по-другому; здесь она соответствует определениям, используемым в промышленных системах проектирования БД.

- **СУБД** (система управления базой данных). Компьютерная программа для управления данными и доступа к ним. Во всех промышленных системах прикладные программы для работы с данными не имеют возможности обратиться к данным БД иначе как через СУБД.

Моделированием (например, составлением карт местности) человечество занимается не одну тысячу лет, и современные базы данных лишь переводят эту деятельность в компьютерную область². Но современное моделирование средствами БД наследует из далекого прошлого и несколько общих проблем. Например:

- Модель и предметная область (объект моделирования) по определению не совпадают. Степень учета в модели обстоятельств предметной области целиком определяется проектировщиком БД, его опытом, знаниями и умением.
- Отклонение модели от предметной области может оказаться фатальным для использования информационной системы, так как конечный пользователь работает с моделью, психологически полагая, что это и есть предметная область. Но отследить такое отклонение не в состоянии ни одна формальная или же программная система. Ответственность за него опять-таки лежит на проектировщике БД.
- Чем дольше используется модель, тем чаще возникает необходимость ее подправить, чтобы снять возникающие со временем расхождения с изменившейся действительностью.

Последнее обстоятельство (необходимость внести в модель данных изменения) может быть вызвано и субъективными причинами (небрежное начальное проектирование, изменение постановки задачи заказчиком), и объективными (изменение самой моделируемой действительности). Например, в базах личностных данных до 2000-х годов сведения о браке достаточно моделировались парой «муж» — «жена», тогда как с наступлением 21 века все чаще это становится неприемлемым, и старые базы требуется править.

Перечисленные обстоятельства присущи моделированию как таковому вообще, а в случае моделирования с помощью БД задача внесения изменений в модель (а значит в БД) часто становится технологически сложной, затратной и трудоемкой. Из-за этого в жизни перестройку БД сплошь и рядом откладывают «до последнего момента».

Что касается СУБД, то именно эта программа обеспечивает «особо быстрый и удобный способ поиска и извлечения данных», и притом она берет на себя решение этих задач монополично, запрещая прикладным программам обращаться к данным в обход себя.

Относительно терминологии, нередко бытуют вольности словоупотребления. Например, в материалах фирмы Oracle часто говорят о «системе базы данных» (database system), вероятно, подразумевая под этим сосуществующую пару СУБД — БД. В склонном к упрощениям американском английском слово «система» в полном термине порою выпадает, в результате чего пару СУБД — БД часто в обиходе именуют словом database. По той же причине вместо «тип СУБД» часто говорят просто «СУБД», и тогда возникает двусмыслица: СУБД как конкретная работающая программа и СУБД как конкретный набор программного обеспечения для обслуживания доступа приложений к данным. Это не исключение: подобная многосмыслица присуща понятию «модель данных», о чем будет сказано ниже, и многим другим понятиям как в базах данных в частности, так и в информационных технологиях вообще. Иногда в этом ничего страшного нет, и можно сориентироваться по контексту употребления, но иногда такие вольности приводят к туману в понимании и в выражении мыслей.

1.1.2. СУБД

СУБД обеспечивает работу приложений, а, в конечном счете, пользователей, с информационной моделью. Основное назначение СУБД состоит в делегировании управления данными от прикладной программы одной специальной программной системе, которая вне зависимости от того, какая прикладная программа или же какой пользователь работает с данными, единым во всех случаях образом:

² Считается, что человеческий мозг отличается от мозга животных именно способностью моделировать, имеющей целью предсказание последствий собственных действий.

- защищает данные от рассогласованности,
- оптимизирует выполнение операций с данным,
- оптимизирует обращение к данным,
- выполняет прочие необходимые действия.

В число функций, которые обеспечиваются современными СУБД, входят следующие.

- Поддержка логической модели данных (определение данных, оперирование данными).
- Восстановление данных (транзакции, журнализация, контрольные точки).
- Управление одновременным доступом к данным в БД.
- Безопасность данных (права доступа и прочее).
- Самостоятельная оптимизация выполнения операций.
- Прочие, в том числе вытекающие из перечисленных (администрирование, статистика, распределение данных и т.д.).

1.1.3. Реляционный подход к моделированию данных

Наиболее существенное влияние на современные промышленные виды СУБД, включая Oracle, оказал реляционный подход к моделированию данных. Он основывается на использовании «реляционной теории», частью которой является «реляционная модель». Последняя берет свое начало со статьи своего основателя, Э. Кодда (E. F. Codd), *Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks*, опубликованной в IBM Research Report RJ599 в 1969 году. Впоследствии реляционная модель пережила всплеск интенсивного изучения и уточнения широким кругом специалистов, а в настоящее время она развивается главным образом усилиями К. Дейта (C. J. Date), сподвижника и коллеги Кодда во времена создания модели.

1.1.3.1. Реляционная модель данных

*Все, что я съел, и все, что я выпил, осталось со мною;
Все остальное, что есть, право, не стоит щелчка.
Надпись на могильной плите Сарданапала
(Ашшурбанапала)*

Выражение «модель данных» часто понимается в двух разных смыслах: как формальное описание некоторой конкретной предметной области и как инструмент для составления подобных описаний. Нужный смысл обычно приходится определять по контексту. Здесь выражение «реляционная модель данных» понимается как инструмент составления в БД конкретных описаний конкретных предметных областей.

Реляционная модель при необходимости может быть описана математическим языком, то есть наиболее точным из изобретенных человеком. Ниже приводятся нестрогие определения некоторых понятий реляционной модели.

- «Тип данных» (type) — множество допустимых величин («область определения») и операций. Для всех типов существуют операции сравнения и присвоения. Величинам не запрещено иметь структуру, например, объекта.
- «Отношение» (relation) — множество атрибутов: уникальных имен с уточнением типа данных; плюс множество «наборов величин» («фрядов»), соответствующих атрибутам. Величины в наборах могут быть представлены только единичными значениями соответствующих атрибутам типов, то есть быть скалярами («1-я нормальная форма»).
- «Переменная отношения» (relation variable) — *переменная* типа отношения конкретного вида, необходимое понятие для определения в базе данных действий по «обновлению отношений», «внесения изменений в данные». В нарушение точности и в силу поверхностно-ознакомительного характера настоящего материала далее вместо названия «переменная отношения» будет употребляться

просто «отношение» (подобно вольному употреблению слова «целое» вместо «переменная целого типа»).

- «Ключ» (key) — группа атрибутов, величины для которых во всех наборах в отношении различны, но ни одна подгруппа этих атрибутов таким свойством уже не обладает (свойство «минимальности» ключа). В частности, группа может состоять из единственного атрибута. Ключ в отношении обязан иметься всегда, а если их несколько, один из них обязан быть назначен «первичным» (primary).
- «Внешний ключ» (foreign key) — группа атрибутов, значения которых в каждом наборе величин отношения обязаны совпадать с набором величин какого-нибудь ключа. Внешние ключи в отношении не обязательны и провозглашаются по потребностям моделирования.
- «Операции» (operation) — множество общих действий над отношениями, дающих в результате опять-таки отношения («замкнутость операций»). Используются для получения новых отношений в нуждах последующего моделирования или при извлечении из базы нужных данных. Перечень операций можно определять по-разному; в первых предложениях модели приводилось восемь операций (проекция, соединения, отбора и пр.), уже не минимальный набор, как компромисс между отсутствием избыточности и удобством употребления.
- «Реляционная база данных» (relational database) — набор отношений.

«Тип данных» иногда называют «доменом» (domain; но иногда под «доменом» понимают только «область определения» величин). «Набор величин» (tuple) по-русски иначе называют «кортежем» или «*n*-кой».

Для удобства отношения часто изображают в виде таблиц, хотя такое представление неправомерно (в отношении не определен ни порядок атрибутов, ни порядок наборов величин, в отличие от таблицы). В SQL, на основе которого построена в том числе СУБД Oracle, понятие «отношения» (а точнее, понятие «переменной типа отношения») как инструмента моделирования заменено как раз на «таблицу». Другим представлением данных отношения может быть гиперкуб, и к нему тоже иногда удобно прибегать в рассуждениях об имеющейся БД.

Если отказаться от определительного слова-кальки «реляционный», то термин «реляционная БД» можно перевести как «БД отношений» (точнее, «БД построенная *посредством* отношений»; отношений как инструмента, а не объекта моделирования: иначе исходный термин был бы relation database). Точно так же термин «реляционная модель» можно перевести как «модель отношений», то есть «система понятий для построения модели предметной области в виде набора отношений». По ряду причин, в том числе исторического и языкового характеров, этого не было в свое время сделано.

Все взаимоотношения данных описываются *явно и только* величинами в наборах (в других подходах к моделированию может быть иначе). Никаких «подразумеваемых» зависимостей (в том числе на уровне программной логики), кроме сформулированных переменными отношений, нет. Реляционный подход разграничивает описание данных и сопутствующую приложению программную логику (в противовес, например, объектному подходу).

Приведенный взгляд на реляционную БД (набор отношений и операции) характерен для *реляционной алгебры*. Это не единственная точка зрения. Каждый набор величин в переменной отношения можно понимать как истинное высказывание («предикат»): имеется такой-то сотрудник с такими-то свойствами; такой-то отдел и так далее. Тем самым реляционная база данных в каждый момент времени представляет собой набор истинных высказываний о предметной области, сформулированный через отношения. По сути, набор высказываний в переменных отношений и образует модель предметной области, представленную базой данных. Такой взгляд на реляционную БД характерен для *реляционного исчисления*. Оба взгляда на реляционную модель хорошо изучены и доказана их выразительная равносильность.

1.1.3.2. Проектирование реляционной базы данных

Задачей проектирования реляционной базы данных можно считать достижение такой системы отношений, которая воспроизводила бы в БД необходимые утверждения о предметной области, и при этом все сведения о предметной области были бы представлены в БД однократно, не повторялись. Такое проектирование, в отличие от реляционной модели, не поддается математическому описанию и в значительной мере определяется здравым смыслом и опытом разработчика.

Существующая теория проектирования реляционной БД не учит, как нужно строить БД. Вместо этого она учит тому, какими неприятностями чревато «неправильное» проектирование: ее иногда называют «хорошим источником плохих примеров».

Устранение избыточности

Значительная часть усилий по проектированию реляционной БД связана с устранением избыточности. Избыточность провоцирует «аномальности обновлений» данных, в результате которых формально правильно составленные вопросы к БД смогут выдавать неверные данные. К сказанному есть два важных замечания. Во-первых, избыточность тут подразумевается применительно к логическому описанию данных, в то время как избыточность физического хранения может быть оправдана и разумна. Во-вторых, устранение избыточности, будучи необходимым для «правильного» построения БД, само по себе не гарантирует правильности моделирования предметной области.

Простой пример устранения избыточности показан ниже.

Пусть в отношении, представляющем сведения о сотрудниках, есть атрибуты «зарплата», «комиссионные» и «доход». Если по правилам моделируемой предметной области доход сотрудника складывается исключительно из его зарплаты и комиссионных, один из перечисленных атрибутов следует из определения отношения убрать — скорее всего это будет «доход»:

Сотрудники

...	Зарплата	Комиссионные	Доход
...	1600	300	1900
...	1250	500	1750



Сотрудники

...	Зарплата	Комиссионные
...	1600	300
...	1250	500

X

В других случаях устранение избыточности может выглядеть отнюдь не столь очевидно. Представьте, что в БД требуется хранить почтовый адрес, включая индекс. Часто для этого используют отдельные атрибуты для индекса и прочих частей адреса, таких как город, улица и так далее. Однако подобное хранение, строго говоря избыточно, так как почтовый индекс однозначно определяется «словесной частью» адреса. Двойное хранение сведений будет держать открытой лазейку для появления содержательно неверных данных в БД, но чтобы воспрепятствовать этому, потребуется заметно усложнить схему и утяжелить БД (что же, идти на хранение не нужного для иной цели соответствия «словесной части» адреса почтовому индексу ?) и как следствие — усложнить и замедлить, казалось бы, простые обращения к БД. В силу таких издержек разработчик в жизни нередко предпочитает простоту организации данных рискам, связанным с избыточностью (или, если удастся, снимает проблему рассогласования данных с помощью ограничений целостности либо триггерных процедур).

Хотя в общем борьба с избыточностью данных в БД — процесс неформализованный, известно две техники, позволяющие устранять избыточность и доведенные до математического уровня описания: нормализация и ортогонализация. Ниже приводится их нестрогое ознакомительное изложение.

Нормализация реляционной базы данных

Нормализация есть приведение отношения к «нормальному виду» путем дробления его на несколько других, связанных внешними ключами. Дробление осуществляется построением проекций исходного отношения, однако задача состоит в том, чтобы обратное соединение полученных в результате дробления отношений возвращало бы нас к исходному отношению, то есть чтобы такое дробление происходило без потерь (искажения) информации.

Наиболее популярная схема подобного «правильного» дробления состоит в устранении из отношения лишних функциональных зависимостей. Данные в отношении функционально зависят друг от друга, если одни из них могут быть определены через другие.

Пусть в отношении с данными о сотрудниках присутствуют и номер отдела сотрудника, и название отдела. Чтобы устранить зависимость между номером и названием отдела в отношении «сотрудники», достаточно оставить в нем только один из двух атрибутов (скорее всего это будет «номер отдела»), создать отношение «отделы» с атрибутами «номер отдела» и «название», объявить в нем «номер отдела» ключом. «Номер отдела» в «сотрудниках» следует объявить внешним ключом, связав его с «номером отдела» из «отделов»:

Сотрудники

...	Smith	20	Research
...	Allen	30	Sales
...	Ward	30	Sales



Сотрудники

...	Smith	20
...	Allen	30
...	Ward	30

внешний ключ

ключ

Отделы

...	20	Research
...	30	Sales

Избавляться от подобного рода функциональных зависимостей в отношениях можно, пока они не окажутся приведенными к «нормальной форме BCNF», Бойса/Кодда (Boyce/Codd; по праву первенства ее следовало бы назвать «нормальной формой Хита», Heath). В таких отношениях, образно выражаясь, «каждое сведение относится к ключу, всему ключу (со всеми его атрибутами) и только к ключу» (в оригинальной фразе на английском вместо «сведения» дословно сказано «факт») с тем добавлением, что ключей может быть несколько. Иными словами, в отношении, удовлетворяющем BCNF, произвол величин в наборах ограничен только правилами ключа, возможно внешнего ключа и типов атрибутов. Несколько менее строгий вариант BCNF именуется «3-ей нормальной формой».

Если ключи в отношениях состоят из групп атрибутов (не являются «простыми»), то для избавления от лишних функциональных зависимостей данных дополнительно потребуется привести отношения к «пятой нормальной форме». Если же для дробления рассмотреть некоторые обобщенные версии операций проекции и соединения, то вдобавок потребуется привести отношения к «шестой нормальной форме». Таким образом, на практике получить выгоды от нормализации проще всего, имея дело с простыми (одноатрибутными) ключами в отношениях.

Все предложенные нормальные формы линейно упорядочены, так что достижение в отношении какой-нибудь из них по определению означает выполнение ряда «предыдущих».

Нормализация отношений способствует:

- устранению избыточности данных в БД;
- как следствие, избавлению от некоторых аномалий обновления данных;
- упрощению изменения, а иногда и запросов к данным;
- упрощению формулирования ограничений целостности;
- представлению данных предметной области в БД в более естественном виде.

Часто процесс устранения избыточности автоматически влечет за собой прочие перечисленные выгоды. Приведение отношений к нормальным формам не способно устранить все виды избыточности, но считается действенным средством для достижения этой цели. Иногда нормализацию данных называют формализованным проявлением здравого смысла. В то же время:

- механическое сведение к нормальной форме BCNF в случае составных ключей может оказаться неоправданным и вступить в противоречие с необходимостью сохранить в модели определенную долю избыточности;
- сохранение избыточности данных, в том числе вследствие недонормализованности отношений, способно приводить к аномалиям обновлений;
- нормализованные данные может оказаться сложнее изменять;
- нормализация неоднозначна и часто допускает разные способы дробления таблиц;
- нормализация не решает всех проблем моделирования, заставляя разработчика БД прибегать к дополнительным правилам ограничения целостности данных.

В жизни в SQL-системах нормализация (применительно к таблицам) часто не соблюдается в угоду ожиданию скорости доступа, простоты изменения данных и их представления. Для обозначения намеренной такой практики даже используется особый термин: «денормализация». Иногда подобные ожидания оправдываются, однако в качестве оборотной стороны это порождает риски расхождения модели в БД с предметной областью, а то и некорректности используемой модели. Некоторые эксперты полагают, что выигрыш, который в определенных обстоятельствах способна дать денормализация, — мифический.

Ортогонализация отношений

В то время, как нормализация отношений ставит своей целью избавление от избыточности в пределах отдельных отношений, ортогонализация данных пытается избавиться от повторений общих данных в разных отношениях. Тем самым она дополняет нормализацию.

Упрощенный вариант принципа ортогонализации данных утверждает, что один и тот же набор величин не имеет право повторяться в разных отношениях. Например, с точки зрения этого принципа желательно следующее преобразование отношений в БД:

Сотрудники 20

Номер	Имя	Зарплата	...
7369	Smith	900	...

Сотрудники 30

Номер	Имя	Зарплата	...
7499	Allen	1600	...
7521	Ward	1250	...



Сотрудники

Номер	Имя	Зарплата	...	Отдел
7499	Allen	1600	...	30
7521	Ward	1250	...	30
7369	Smith	900	...	20

Существует и более сложная формулировка принципа ортогональности для проектируемых отношений, требующая для иллюстрации менее очевидных примеров.

Формально идеальная база данных отношений

Таким образом, с формальной точки зрения идеальной реляционной БД можно полагать такую, в отношениях которой отсутствует избыточность данных, а единственным видом зависимости в отношениях являются ключи и внешние ключи. Возможности установления значений неключевым атрибутам определяются исключительно типами атрибутов.

Формально идеальное построение БД обеспечивает качество ее эксплуатации. Тем не менее:

- (а) такое построение не всегда дается легко (например, за счет перегрузки базы обилием взаимосвязанных отношений/таблиц), и часто сознательно, или бессознательно нарушается (например, употреблением «правил целостности») в угоду простоты схемы или эффективности;
- (б) формально правильное проектирование БД почти не решает задачу соответствия моделируемой действительности, лежащую в другой плоскости (смотри выше).

Сказанное распространяется и на базы данных, основанные на SQL, с точностью до терминологии, принятой в этом языке.

1.1.4. Программное воплощение реляционной СУБД

Среди промышленных систем, наиболее распространенных на рынке, ближе всего к реляционным подошли системы, основанные на SQL. К ним относится Oracle. Все такие системы наследуют у реляционного моделирования важные свойства. Например:

- в БД, основанных на SQL, все данные хранятся в таблицах и только в таблицах (аналог переменных типа отношения в реляционной модели);
- в таблицах допускается наличие первичных ключей, уникальности для столбцов (аналог ключей) и внешних ключей;
- запросы и команды изменения данных формулируются преимущественно не процедурно (алгоритмически), а заявительно, путем перечисления свойств ожидаемого от БД результата.

Упомянутые понятия SQL (таблицы, уникальные столбцы, внешние ключи) являются неточными параллелями понятий реляционной модели (отношения, ключи, «реляционные» внешние ключи соответственно), однако же имеют их своими прообразами и на деле воспроизводят некоторые их важные качества.

Наработки по проектированию «баз данных отношений» в целом переносятся на системы типа SQL. В частности, для БД типа SQL также желательны устранение избыточности, нормализация и ортогонализация, но уже таблиц, а не отношений.

В то же время все системы на основе SQL имеют отличия от реляционных, как существенные, так и не очень, и в конечном итоге их не правомерно причислить к разряду реляционных:

- ключей (не внешних) в таблице может быть не более одного (так что прилагательное «первичный» в термине «первичный ключ» смысла собою в SQL не добавляет, и всего лишь напоминает об исторической связи этого языка с реляционной моделью);
- в таблицах первичные ключи необязательны, а значит, допускаются повторяющиеся строки (в отношениях наборы величин суть истинные высказывания о предметной области, а повторение высказывания многократно никак не «улучшает» его истинности; но также имеются и другие, технические побудительные причины к неприятию повторений);
- понятие реляционных «доменов» (типов данных) недореализовано или реализовано с ошибками (встроенные типы данных примитивны и нарушают строгую типизацию, а объектные типы не обязаны поддерживать сравнение);
- требуется «ручная» оптимизация запросов (а полагалось, что формулировка запроса выбирается программистом из соображения удобства восприятия, а оптимизация выполнения — забота исключительно СУБД);
- другие.

Неприятные практические последствия подобных отличий и противоречий описаны в литературе.

Попытки создать промышленную реляционную СУБД за всю историю существования реляционной теории не увенчались успехом. Они не прекращаются (http://en.wikipedia.org/wiki/Data_language_specification#Tutorial_D), но об использовании SQL в таких экспериментальных системах речи не идет.

1.1.5. Другие подходы к моделированию данных и другие типы СУБД

Наряду с реляционным подходом к моделированию данных существуют и другие, обладающие разной степенью строгостью определения, распространенности и разными областями применения.

- Объектный подход.
- Объектно-реляционный подход.
- Многомерное моделирование.
- Дедуктивное моделирование.

Объектный подход к моделированию БД дополняет объектно-ориентированное программирование и имеет давнюю историю. В число основных понятий такого подхода входят следующие: объект, класс/тип, метод/оператор/функция, сообщения; инкапсуляция, уникальный идентификатор объекта, наследование, полиморфизм операций. Этот набор неоднозначен и сложен для начального освоения разработчиками, что отрицательно сказалось на распространении объектного подхода. Вторым отрицательным фактором является отсутствие методологии проектирования объектных БД; третьим — технические трудности реализации, вызванные необходимостью долговременного хранения объектов. Последним объектное моделирование в БД решительно отличается от объектного подхода в программировании, где отсутствует задача хранения объектов произвольно продолжительное время.

Объектно-реляционный подход к моделированию БД еще менее отчетливо сформулирован, нежели чисто объектный. Практически он обернулся включением (во второй половине 90-х годов) в состав ведущих систем управления данными на основе SQL возможностей хранения объектных данных. Некоторые основы такого включения (например, приравнивание класса домену, или же объекта строке таблицы) остаются спорными. Фирма Oracle расширила табличные средства моделирования данных объектными возможностями в версии 8 своей СУБД, которую с этого времени стала именовать «объектно-реляционной». Поверхностное знакомство с объектными возможностями Oracle состоится в соответствующем разделе далее. Более плотное знакомство требует заметного увеличения объема материала.

Многомерное моделирование в базах данных имеет еще более давнюю, чем объектное, историю, уходящую в 1970-е годы. К его основным понятиям могут быть причислены: гиперкуб данных (или, по многомерной терминологии, «фактов»), измерения, иерархии, атрибуты. По большому счету (теоретически), оно не добавляет нового к тому, на что способны табличные СУБД, однако предлагает собственное представление данных и операции, что делает его удобным, а при соответствующей организации и эффективным в системах анализа данных. Фирма Oracle, купив в конце 1990-х годов систему управления многомерными данными, встроила в свою СУБД «машину OLAP» (on-line analytical processing), основанную на функциональности купленной системы. С версии 9 средства OLAP поставляются в составе Oracle Enterprise Edition. Они воплощены в двух исполнениях: унаследованном, предполагающем встроенную в СУБД самостоятельную систему хранения, и со сведением данных гиперкуба к привычной таблице. Последнее исполнение рассчитано на работу с данными посредством SQL, слегка расширенного для этого случая фирмой-изготовителем. Это расширение специфично, несколько непоследовательно и далее не рассматривается.

Перечисленные подходы к моделированию данных существенно отличаются от реляционного тем, что не имеют математического обоснования, с вытекающими из этого последствиями:

- отсутствием общепризнанной и экономной системы понятий,
- невозможностью оценки создаваемых БД аналитическими средствами,
- отсутствием методологии проектирования БД.

Из-за этого распространенность перечисленных подходов на практике ограничена.

Дедуктивный подход позволяет хранить в БД «факты» и «правила вывода» и получать на их основе новые факты. Он позволяет обрабатывать рекурсию и получать ответы на вопросы, в принципе не доступные прочим видам современных СУБД. Однако с его разработкой и программным воплощением связаны существенные трудности, и ни одной промышленной дедуктивной СУБД пока не известно.

Помимо указанного, можно говорить о:

- безмодельных БД;
- слабоструктурированных БД.

Примером первых можно привести документальные базы текстовых документов, где связи между документами не предписываются моделью данных, а выясняются (вычисляются) в результате обработки поступающих на хранение документов. СУБД Oracle поддерживает такое хранение штатным образом в рамках своей обычной БД, начиная с версии 7.3. Эта возможность ныне носит название Oracle Text и поддерживается на двух уровнях: SQL и программном (обращением ко встроенным в БД пакетам). В силу своей специфики и большого объема материала эта тема ниже не затронута.

Примером слабоструктурированных данных можно привести БД документов XML. Хранение документов XML в БД Oracle разрешила в версии 9.2 в двух вариантах: основном и продвинутом. Продвинутый вариант получил название XML DB, воплощая собою своего рода «базу (XML) в базе (обычной)». Как и Oracle Text, XML DB поддерживается взаимодополняюще средствами SQL и встроенных программных пакетов, и по тем же причинам в этом материале она освещения не получила.

1.2. SQL

1.2.1. Что такое SQL ?

Structured Query Language — язык, первоначально задумывавшийся как инструмент работы с реляционными базами данных, но с этой ролью в конце концов не справившийся.

Состоит из нескольких «подязыков» разной функциональной направленности.

- Запросы к данным БД (query language, QL) — выборка данных из таблиц предложением SELECT.
- Язык манипулирования данными («обработки данных»; ЯМД; data manipulation language, DML) — изменение данных в существующих таблицах.
- Язык управления транзакциями — например, фиксация или отмена произведенных программой изменений в БД.
- Язык определения данных (ЯОД; data definition language, DDL) — например, создание таблиц и прочих объектов хранения или изменение их свойств.
- Язык управления данными (ЯУД; data control language, DCL) — управление разрешением доступа к данным путем выдачи или изъятия полномочий командами GRANT и REVOKE, заданием в системе пользователей данных.

Единицами языка являются «команды», они же «предложения» (commands и statements), они же «операторы». Слово «запрос» (query) иногда трактуют расширительно, понимая под ним любую команду SQL как «запрос к БД». С другой стороны предложения SELECT часто обсуждают вместе с предложениями категории DML и в таких случаях иногда причисляют к последним.

Общие структурные свойства предложений DML и запросов можно охарактеризовать следующим образом.

- *Непроцедурность*. Проявляется двояко.
 - Каждое предложение SQL является самодостаточным; предложения SQL не предназначены для процедурного описания действий.
 - Считается, что запрос к данным в SQL формулируется не указанием процедуры вычисления результата, а описанием свойств результата (statement буквально «утверждение»); СУБД же по

указанным программистом свойствам автоматически строит последовательность действий, приводящую к требуемому результату. Современный SQL содержит некоторые отклонения от такой непроцедурности.

- **Множественность.** Запросы SQL на выборку и изменение данных есть операции со множествами строк, не предполагающими упорядоченности.

По способу использования предложения SQL разделяются на две разновидности.

- **Диалоговый SQL.** Вариант формулировок предложений языка (на выборку и изменение данных), предполагающий диалог программиста с СУБД по схеме «вопрос-ответ». Пример — работа с БД через программу Oracle SQL*Plus.
- **Встроенный SQL.** Вариант формулировок языка, предполагающий размещение предложений SQL в программе (на C, C++, Cobol, Java, PL/SQL и проч.).

Вторая разновидность — встроенный SQL — фактически превращает SQL в язык программирования БД. Именно в таком качестве он сейчас главным образом используется, невзирая на то, что задумывался когда-то языком для конечного пользователя. Похожие изменения жизнью замыслов создателей наблюдаются и в других знаменитых продуктах информационных технологий, например, в Java.

1.2.2. История и стандарты

Первая версия SQL была предложена в 1973 году фирмой IBM под названием *Sequel*. В 1980 году этой же фирмой язык был воплощен уже под нынешним названием в проекте System R по построению прототипа реляционной СУБД (неформальную персонализированную историю и обсуждение последствий создания смотри в <http://www.citforum.ru/database/digest/sql1.shtml>). Успех System R дал начало многочисленным попыткам разных фирм воспроизвести эту систему, а вместе с ней SQL. Первый коммерческий результат такой деятельности принадлежит нынешней фирме Oracle. (Формулировки некоторых команд SQL в Oracle до сих пор напоминают о System R фирмы IBM).

Для борьбы с несовместимостью воплощений SQL в разных системах язык вскоре оказался вовлечен в активную деятельность по стандартизации, проводившуюся разными организациями в разное время: в том числе X/Open Group, SQL Access Group, FIPS, Госстандарт РФ. Сейчас действуют два комитета по стандартизации SQL: ANSI (в США) и ISO (международный), работающие синхронно.

Основные вехи стандартизации перечислены ниже:

- SQL-86 (1986/1987 гг.);
- SQL-89 (1989 г.; фактически SQL-86 с малыми добавками);
- SQL-92 (синонимы: SQL2, SQL92; 1992 г.)
 - Вводный уровень (Entry Level)
 - Переходный уровень (Transitional Level)
 - Промежуточный уровень (Intermediate Level)
 - Полный уровень (Full Compliance);
- SQL:1999 (синонимы: SQL3, SQL-95, SQL1999, SQL-99, SQL99; главные дополнения — объектные возможности языка, хранимые процедуры)
 - Основной уровень (Core)
 - Расширенный уровень (Optional);
- SQL:2003 (пример дополнений — возможности XML). С этого момента очередные версии стандарта выходят не в полном объеме, как прежде, а в виде «дополнительных частей»;
- SQL:2006 (развивает расширения, касающиеся XML, возникшие в SQL:2003);
- SQL:2008 (ряд частных дополнений, как например TRUNCATE TABLE или выдача первых N записей).
- SQL:2011 (темпоральные БД, конвейерные операции DML, другие новшества).

На основе SQL-92 построен ГОСТ Р ИСО/МЭК 9075-93.

Максимально достигнутый промышленными поставщиками уровень соответствия стандарту в настоящее время — Core SQL:2008, да и то с отклонениями. Он является продолжением (с расширениями и поправками) Core SQL:1999, а тот — продолжением Entry Level SQL-92.

Исторически SQL — не единственный предлагавшийся язык доступа к БД, а по мнению некоторых специалистов — и не лучший. Стандартизация SQL — одна из причин, по которой именно он получил широкое признание. Теоретически стандартизация обеспечивает независимость прикладных программ от типа используемой СУБД. Практически же каждый разработчик СУБД, включая фирму Oracle, предлагает не чисто стандартную реализацию, а собственный диалект SQL.

Противоречивость процессов стандартизации состоит в том, что коммерческие фирмы-разработчики склонны в первую очередь к собственным решениям, и только во вторую заинтересованы в общих; потребители же в первую очередь заинтересованы в общих решениях, и только во вторую — в частных. Росту диалектов способствует отсутствие в стандартах SQL однозначных предложений для некоторых свойств, например:

- выдача даты и времени суток,
- порождение значений для искусственных ключей,
- семантика некоторых типов данных.

Далее по тексту ссылки на «стандарт SQL» и «стандарт ANSI/ISO» относятся фактически к стандартам SQL:20xx. Ссылки на SQL-92, SQL:1999 и SQL:2003/8 уточняют время появления упоминаемого свойства стандарта SQL:20xx последней версии.

Исходное название — Sequel — происходит как сокращение от *structured English query language* (так что некоторые ветераны программирования по сей поре произносят SQL как «сиквел»). Слово *english* в полном названии неслучайно: нынешний SQL замысливался в виде такого подмножества естественного языка, на котором потребитель данных мог бы обращаться к БД. Со временем метаморфоза произошла с «потребителем». Раньше полагали, что им будет конечный пользователь, а в наше время им оказался программист. Теперешний конечный пользователь как правило ничего не знает об SQL, работая с БД средствами графики или более высокого (по крайней мере архитектурно) уровня, построенными с помощью SQL, но уже программистом. Тем не менее, как и раньше, многие предложения на SQL действительно напоминают обычные человеческие высказывания. В этом таится большая опасность для программиста, так как на деле SQL — формальный язык (хоть и не самый сложный), который от человеческого отделяет пропасть. Выписывая предложение на SQL, программист нередко подсознательно пытается осмыслить написанное по-человечески, что иногда не страшно, но чаще совершенно искажает совершаемое СУБД на деле. Можно привести массу примеров, когда по-человечески воспринимаемая запись на SQL в действительности означает совсем иное. Заложенная когда-то в Sequel близость к формулировкам на английском языке вынуждает сегодняшнего программиста на SQL все время быть начеку. Показательно, что в расшифровке SQL (*structured query language*) слово English выпало, но инерция развития языка сохраняется.

1.2.3. Диалект SQL в СУБД Oracle

СУБД Oracle³ версии 1 (под другим названием) появилась в 1978 г. и была написана на языке ассемблера для PDP-11 под ОС RSX. Oracle версии 2 появилась в 1979 г. и стала «первой коммерческой реляционной СУБД с использованием SQL», а в версии 3 ее ядро было переписано на языке C. С версии 8 фирма Oracle называет свою СУБД «объектно-реляционной». Фактически же это была и остается SQL-ориентированная СУБД, использующая собственный диалект SQL.

В силу истории происхождения диалект SQL фирмы Oracle естественно оценивать с трех сторон, определяющих одновременно и факторы влияния на существующий его вид:

- реляционной теории,
- стандартного SQL,
- фирмы-разработчика.

³ В древней Греции оракулом называлось место, где прорицательница-пифия, одурманенная пьянящими парами, как могла сообщала мнение божества по поводу вопроса, интересующего посетителя.

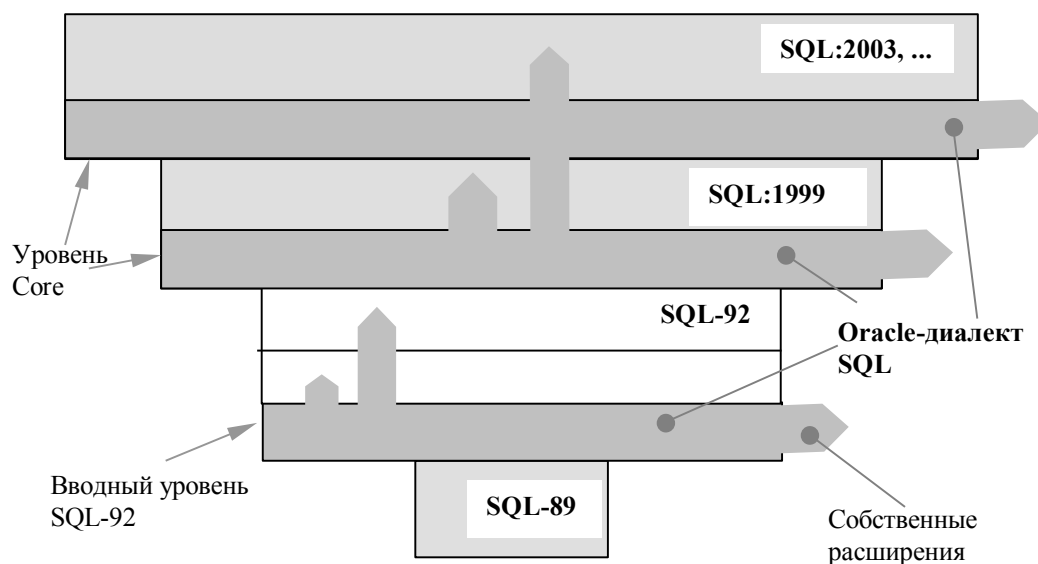
Во всех случаях можно говорить о понимании вопроса соответствующей стороной и об усилиях на проработку того или иного свойства (а в случае фирмы-разработчика — еще и усилий на программирование).

Oracle «в значительной степени» (но не полностью и не в точности) поддерживает уровень Core SQL:2003, некоторые возможности уровня Optional Features SQL:2003 и набор собственных нестандартизованных свойств. Точное перечисление соответствия диалекта SQL Oracle стандарту приводится в документации по СУБД Oracle.

Некоторые собственные свойства диалекта SQL Oracle:

- типы данных
- дополнительные к стандарту функции
- дополнительные конструкции SQL

Объем диалекта SQL в Oracle по отношению к стандартам пояснен следующим рисунком.



Выход за пределы стандарта практикуется не только в Oracle, но всеми без исключения производителями, что девальвирует стандарт.

К этому же можно добавить, что в очередных версиях своей СУБД фирма Oracle все больше и больше новых возможностей (чаще, правда, технологического характера) реализует не посредством SQL в рамках своего диалекта, а программно, с помощью встроенных в БД системных пакетов подпрограмм.

Три понятия — реляционная модель, стандартный язык SQL и диалект SQL в Oracle — безусловно связаны, но связи эти вольные. Стандарт SQL является промышленным стандартом, создаваемым под влиянием реляционной модели, но не воплощающим реляционного языка запросов. Одновременно диалект SQL в Oracle создается для СУБД конкретного типа под влиянием стандарта SQL, но не придерживаясь его как догмы. Стандарт SQL и диалекты SQL всех производителей (включая Oracle) существуют в большой степени самостоятельно, оказывая воздействия друг на друга, но не воспроизводя друг друга в точности.

1.3. PL/SQL

PL/SQL есть процедурный язык, встроенный в СУБД Oracle для возможности работать с хранимой в БД Oracle программной логикой. Наличие процедурного языка предусмотрено стандартом SQL (начиная с

SQL:1999), однако все собственные реализации разработчиков имеют, в отличие от случая с SQL, крайне мало общего между собой и с описаниями стандарта.

Процедурный язык Oracle позволяет создавать в БД триггерные процедуры, моделировать логику предметной области («бизнес-логику»), проявлять самостоятельность БД, например, в виде переноса данных или обращения к интернету, организовывать более сложную, нежели табличную в SQL, защиту доступа.

Для проектировщика БД PL/SQL дополняет SQL и табличную организацию данных. Хотя все данные в БД Oracle хранятся исключительно в виде таблиц, для моделирования действий с данными по правилам предметной области возможностей SQL нередко не хватает. Это случается, например, когда единый с точки зрения приложения составной набор свойств хранится в виде совокупности значений в разных таблицах. Согласованное изменение таких свойств исключительно посредством команд SQL часто неудобно или даже невозможно. Это можно воспринимать как недостаточность языка SQL (хотя последняя отчасти и устраняется объектными возможностями языка). В таких случаях подобные изменения «запаковывают» в подпрограмму, к которой и будет прибегать прикладной разработчик для воздействия на данные. Сама фирма Oracle пользуется этим очень активно, что проявляется в наличии номенклатуры «встроенных пакетов» на PL/SQL, активно приумножающихся числом в каждой очередной версии СУБД. В версии 11 количество только документированных встроенных пакетов превышает 200. Большая часть из них осуществляет изменения данных в служебных таблицах БД через функции и процедуры PL/SQL.

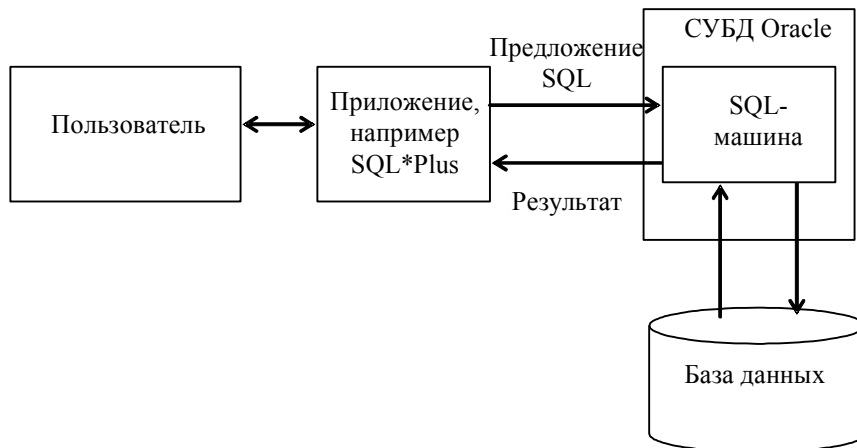
Этим же путем — процедурного изменения данных в БД — часто следуют прикладные разработчики.

Операторы DML и SELECT из SQL могут употребляться на правах предложений языка PL/SQL. Это дает основание фирме Oracle иногда называть PL/SQL «процедурным расширением» SQL. Эффективность обработки СУБД таких операторов выше, чем у поступивших из внешней программы на общеупотребимом языке программирования.

2. Инструменты для общения с базой данных

Фирма Oracle предоставляет два основных инструмента для общения с БД в диалоге посредством SQL: SQL*Plus и SQL Developer. Дальнейшие примеры в тексте, как правило, предполагаются для исполнения в SQL*Plus, однако с разной степенью корректировки исполняемы и в SQL Developer.

SQL*Plus — программа из обычного комплекта ПО Oracle для диалогового общения с БД путем ввода пользователем (или, возможно, из сценарного файла-«скрипта») текстов на SQL и PL/SQL и предъявления на экране компьютера результата, полученного от СУБД:



Запуск SQL*Plus может осуществляться:

- через меню ОС (в Windows);
- из командной строки (во всех ОС).

Пример запуска из командной строки:

```
Command Prompt - sqlplus scott/tiger
C:\Documents and Settings\student>sqlplus scott/tiger
SQL*Plus: Release 10.2.0.3.0 - Production on Thu Jul 9 14:54:03 2009
Copyright (c) 1982, 2006, Oracle. All Rights Reserved.

Connected to:
Oracle Database 11g Enterprise Edition Release 11.1.0.6.0 - Production
With the Partitioning option

SQL>
```

SQL*Plus обрабатывает тексты на трех языках: SQL, PL/SQL и собственном. Во всех случаях регистр набора не имеет значения. Точнее, можно полагать, что, приняв на входе команду на любом из трех языков, «система» (выполняет ли эту работу SQL*Plus или СУБД, в данном случае неважно) повышает регистр всем буквам кроме заковыченных символами ' и ", а потом уж производит обработку (игнорируя лишние пробелы). Следующие две команды SQL содержательно равносильны (выборка всех данных — строк и столбцов — из таблицы DEPT):

```
SELECT * FROM dept;
select * FROM Dept ;
```

Исключение из правила автоматического повышения регистра перед обработкой команды касается значения пароля в версиях, начиная с 11. С этой версии выдача следующих двух команд SQL приведет к установке *разных* значений пароля пользователю SCOTT:

```
ALTER USER scott IDENTIFIED BY tiger;  
ALTER USER scott IDENTIFIED BY Tiger;
```

Символ ; используется в качестве признака окончания ввода команды SQL (они могут быть многострочными), но в некоторых случаях для этого используется / в *первой* позиции новой строки. Символ - используется как перенос продолжения набираемой команды SQL*Plus на новую строку (если эта команда чересчур длинна).

Собственные команды SQL*Plus служат для настроек работы этой программы, установления форматов и выполнения некоторых действий. Их несколько десятков, и полный перечень (в жизни избыточный) приведен в документации по Oracle. Примеры:

- DESCRIBE — выдача на экран общего описания структуры таблиц, представлений данных, типов объектов или пакетов:
DESCRIBE emp
- SET — установка (а SHOW — просмотр) режимов выдачи данных на экран, например установка длины внутреннего буфера формирования строк ответа и установка разбиения ответа на запрос на страницы:
SET LINESIZE 200
SET PAGESIZE 50
- COLUMN — задание формата выдачи данных столбца на экран:
COLUMN object_type FORMAT A20
- CONNECT/DISCONNECT — установление сеанса связи с СУБД, например:
CONNECT scott/tiger

В отличие от SQL и PL/SQL, большинство ключевых слов в языке SQL*Plus имеют сокращенные формы, часто употребляемые в жизни и в литературе, например:

```
DESC emp  
SET LINES 200  
COL object_type FOR A20  
CONN scott/tiger
```

Для обработки вводимых команд SQL (а заодно блоков на PL/SQL) в SQL*Plus применяется внутренний буфер команды. Он обновляется при каждом новом наборе текста на SQL (или блока на PL/SQL). Команда SQL*Plus LIST позволяет выдать на экран текущее содержимое буфера, команда RUN или же символ / — запустить содержимое на исполнение, а команда EDIT — редактировать.

SQL Developer тоже позволяет пользователю обращаться к БД на SQL, но имеет графический интерфейс и графические средства отладки. Для SQL Developer фирмой Oracle и третьими фирмами разработан ряд расширений: для администрирования картографической информации в БД (Oracle Spatial), для графического администрирования средств анализа данных (Oracle Data Mining), другие.

С целью моделирования БД графическим образом можно использовать родственный продукт Oracle SQL Developer Data Modeler.

В качестве графической среды разработки и отладки запросов SQL и программ на PL/SQL имеется также несколько программных продуктов третьих фирм. Они появились раньше, чем SQL Developer, в общем обладают теми же возможностями, но часто более тщательно проработаны в деталях. Дальнейшие примеры можно отрабатывать и с их помощью.

3. Данные для дальнейших примеров

3.1. Таблицы

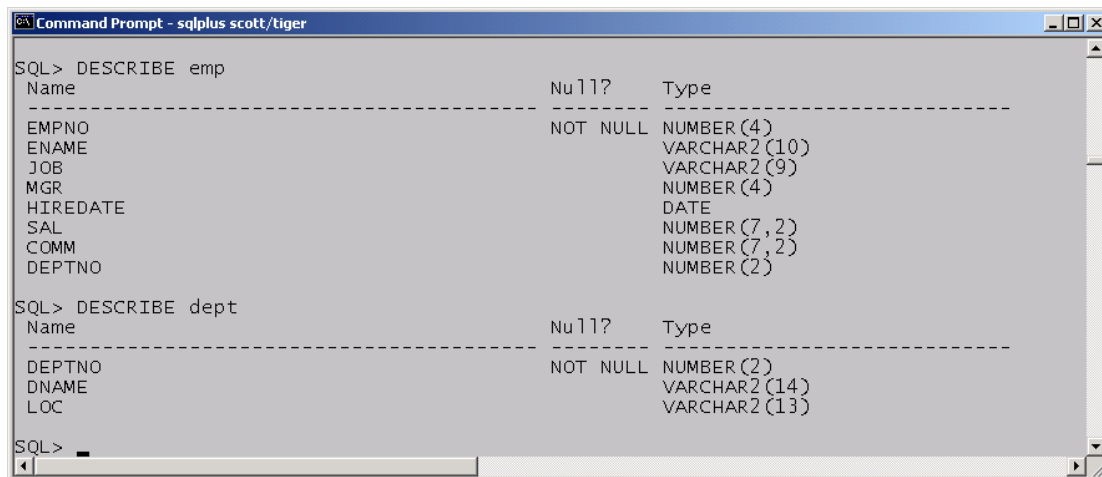
В дальнейших примерах чаще всего будут использоваться две таблицы, хранимые в «схеме данных» с именем SCOTT: это EMP и DEPT.

Потребительски в Oracle и большинстве других подобных систем схемой данных называется специально оформляемое подмножество хранимых объектов, с которыми конкретные прикладные программы имеют право работать в данный момент. Понятие схемы данных задумано ради удобства работы с содержимым БД и защиты доступа. Объектами схемы в Oracle могут быть не только таблицы, но и прочие элементы, такие, как индексы, процедуры и другие (в том числе так называемые «ограничения целостности»). Слово «схема» иногда употребляется и применительно к таблице. Под «схемой таблицы» понимают ее структуру вместе с некоторыми элементами описания.

В стандарте SQL множество рабочих объектов для программы устанавливается несколько сложнее, но тоже с участием понятия «схема». В реляционной модели это понятие не разработано, однако это сделано в общей теории баз данных.

Схема с именем SCOTT поставляется в Oracle с давних времен для демонстрационных целей.

Общее описание таблиц:



```
Command Prompt - sqlplus scott/tiger

SQL> DESCRIBE emp
Name                               Null?    Type
-----
EMPNO                               NOT NULL NUMBER(4)
ENAME                               VARCHAR2(10)
JOB                                  VARCHAR2(9)
MGR                                  NUMBER(4)
HIREDATE                            DATE
SAL                                  NUMBER(7,2)
COMM                                 NUMBER(7,2)
DEPTNO                              NUMBER(2)

SQL> DESCRIBE dept
Name                               Null?    Type
-----
DEPTNO                              NOT NULL NUMBER(2)
DNAME                                VARCHAR2(14)
LOC                                  VARCHAR2(13)

SQL>
```

Данные в таблицах (EMP и DEPT):

```

Command Prompt - sqlplus scott/tiger

SQL> SET LINESIZE 200 PAGESIZE 40
SQL> SELECT * FROM emp;

   EMPNO ENAME      JOB              MGR HIREDATE          SAL        COMM     DEPTNO
-----
   7369 SMITH        CLERK                7902 17-DEC-80           800              20
   7499 ALLEN        SALESMAN             7698 20-FEB-81          1600           300       30
   7521 WARD          SALESMAN             7698 22-FEB-81          1250           500       30
   7566 JONES        MANAGER              7839 02-APR-81          2975              20
   7654 MARTIN      SALESMAN             7698 28-SEP-81          1250          1400       30
   7698 BLAKE        MANAGER              7839 01-MAY-81          2850              30
   7782 CLARK        MANAGER              7839 09-JUN-81          2450              10
   7788 SCOTT       ANALYST              7566 19-APR-87          3000              20
   7839 KING         PRESIDENT            17-NOV-81          5000              10
   7844 TURNER      SALESMAN             7698 08-SEP-81          1500              30
   7876 ADAMS       CLERK                7788 23-MAY-87          1100              20
   7900 JAMES       CLERK                7698 03-DEC-81           950              30
   7902 FORD        ANALYST              7566 03-DEC-81          3000              20
   7934 MILLER      CLERK                7782 23-JAN-82          1300              10

14 rows selected.

SQL> SELECT * FROM dept;

   DEPTNO DNAME          LOC
-----
    10 ACCOUNTING    NEW YORK
    20 RESEARCH      DALLAS
    30 SALES          CHICAGO
    40 OPERATIONS    BOSTON

SQL>

```

Таблица EMP имеет (первичный) ключ: столбец EMPNO; ключ в таблице DEPT — столбец DEPTNO. В таблице EMP столбец DEPTNO образует «внешний ключ», ссылающийся на значения из одноименного столбца таблицы DEPT:

Таблица EMP

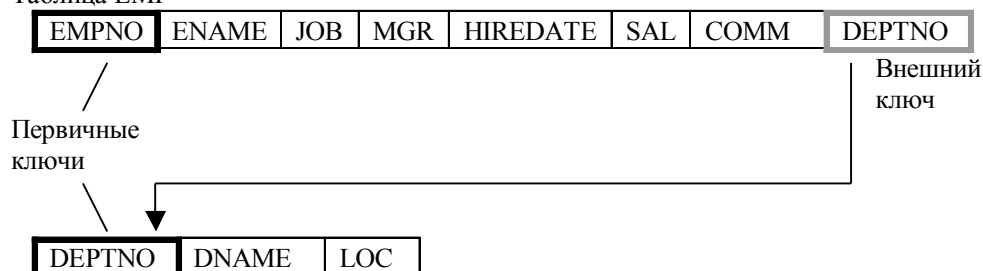


Таблица DEPT

Совпадение имен столбцов внешнего ключа и столбцов адресатов в SQL не обязательно, но в жизни приветствуется.

Таблица DUAL не принадлежит схеме SCOTT, а является «системной» (принадлежит схеме SYS) и доступна для выборки любому пользователю. Состоит из единственной строки и единственного столбца:

```

Command Prompt - sqlplus scott/tiger

SQL> DESCRIBE dual
Name                               Null?    Type
-----
DUMMY                               YES      VARCHAR2(1)

SQL> SELECT * FROM dual;

D
-
X

SQL>

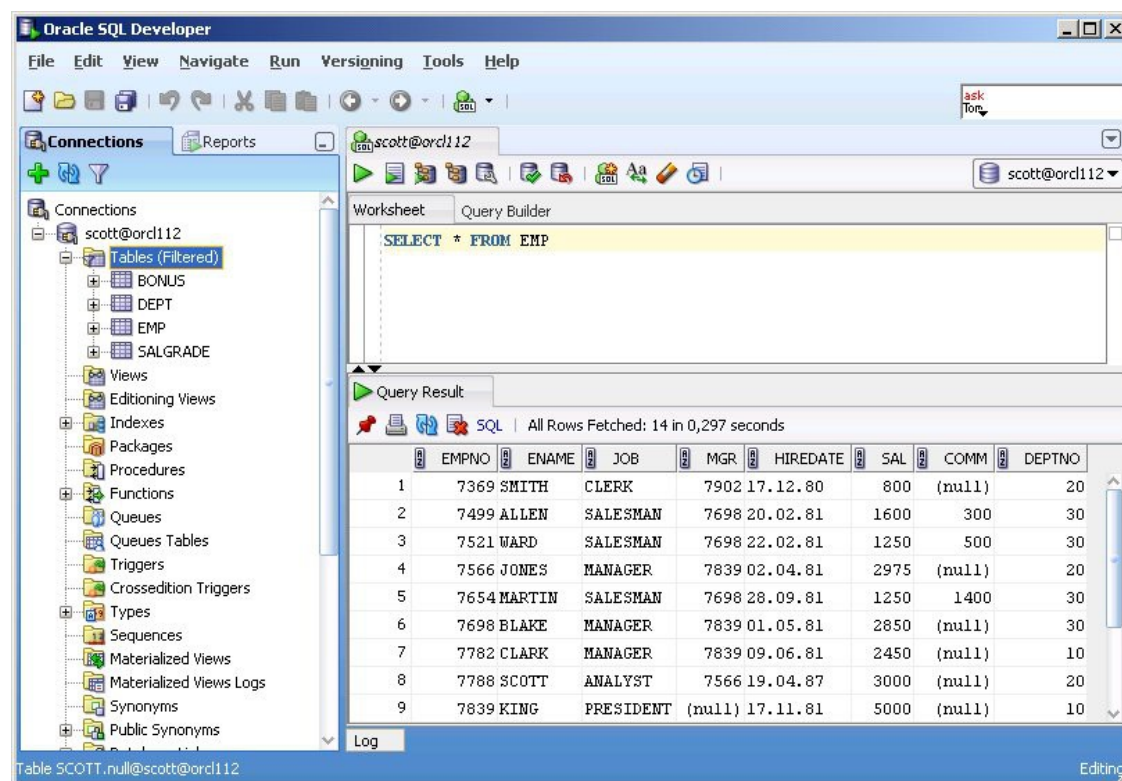
```

Таблица DUAL предназначена играть техническую роль и с этой целью не раз используется в примерах ниже.

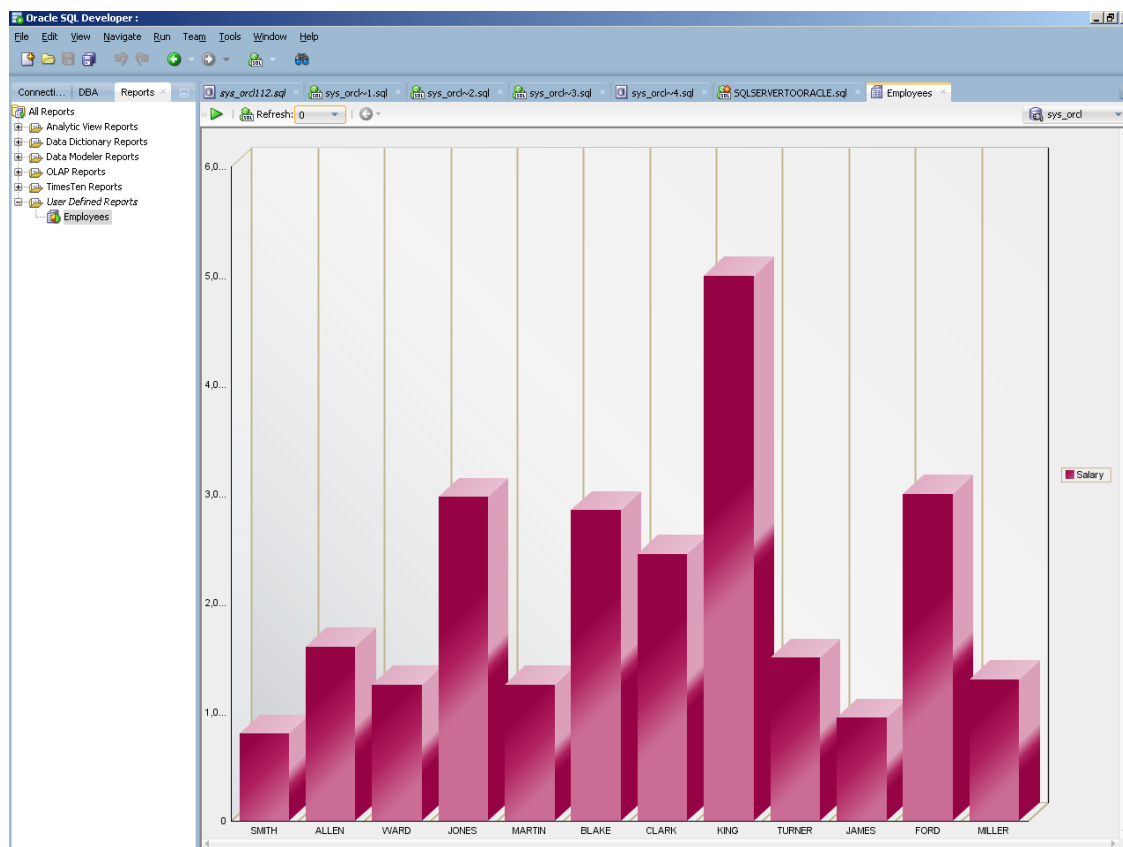
Такой характер ее употребления сложился не сразу. Когда она создавалась в старые времена, то имела две, а не одну, как нынче, строки (отсюда название DUAL) и служила для «удвоения» строк системной таблицы выполнением соединения (join) в запросе о распределении памяти на диске в табличном пространстве БД. Впоследствии эта ее задача оказалась неактуальной, и она приобрела нынешнее значение.

В схеме SCOTT присутствует еще пара таблиц, BONUS и SALGRADE, гораздо менее интересных и далее почти не востребованных.

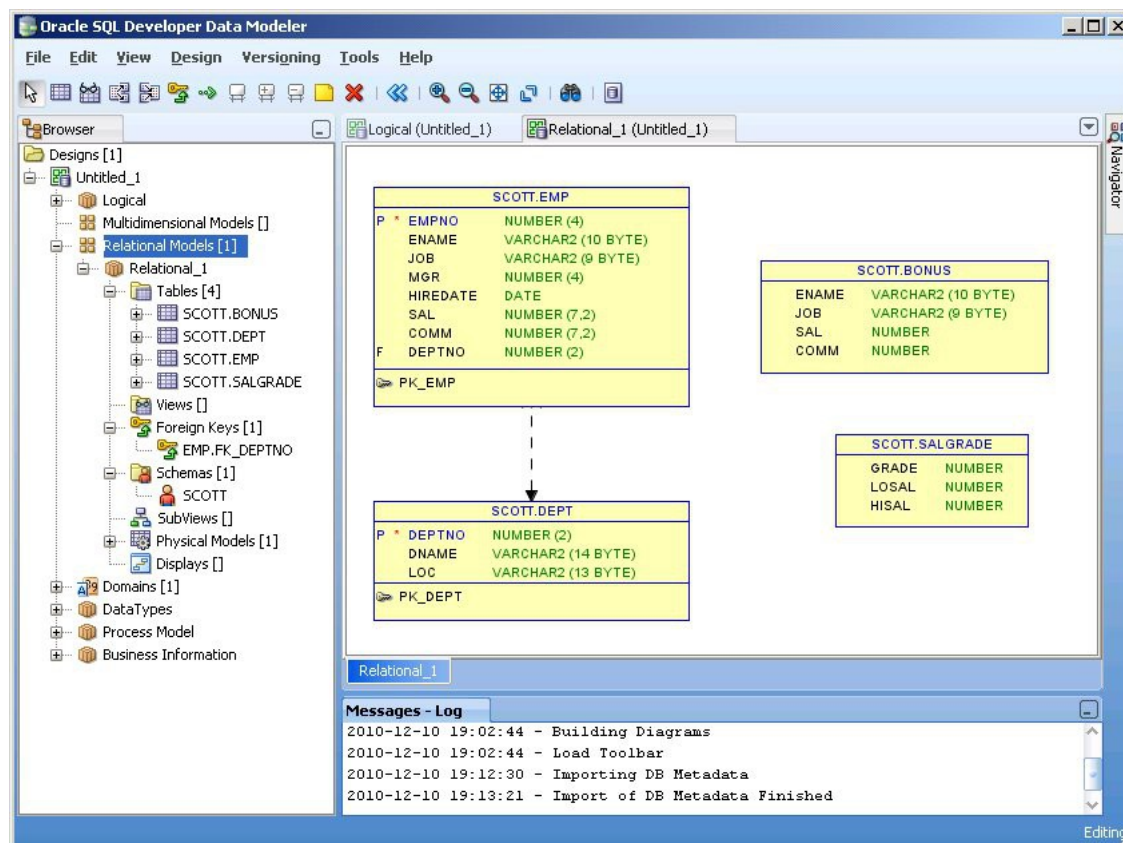
Пример работы с таблицами схемы SCOTT в Oracle SQL Developer:



Пример подачи результата запроса SQL (SELECT ename, 'Salary', sal FROM emp) в Oracle SQL Developer:



Пример работы с таблицами схемы SCOTT в Oracle SQL Developer Data Modeler:



Упражнение. Выяснить в SQL*Plus или SQL Developer структуру (описание столбцов) и содержимое таблиц BONUS и SALGRADE в схеме SCOTT.

Со временем фирма Oracle разработала более правдоподобно и сложно устроенный демонстрационный пример, нежели схема SCOTT, на этот раз из нескольких схем: HR, OE, PM, SH, IX, BI. С ним часто можно столкнуться в нынешних примерах из разных источников. Далее в тексте предпочтение отдано все-таки схеме SCOTT: она небольшого размера, и ее обычно достаточно для решаемых задач.

3.2. Пользователи и полномочия

В отличие от схемы, понятие пользователя в базах данных служит для моделирования возможностей прикладной программы совершать те или иные действия в БД. Что касается действий с объектами БД, то специального разрешения на такие действия Oracle не требует в двух случаях:

- когда объект доступа принадлежит самому пользователю,
- когда действие с объектом объявлено «публичным».

В остальных случаях для выполнения действия пользователь должен иметь специально выданное ему полномочие («привилегию» по терминологии Oracle). Примерами таких полномочий могут служить полномочие создавать таблицу (в собственной схеме!), полномочие обращаться запросом SELECT к чужой таблице.

В БД Oracle самостоятельным является понятие пользователя, а понятие схемы подчинено ему. Так, по команде CREATE USER СУБД создаст в БД *пользователя* и одновременно автоматически — *схему* с тем же именем, к которой будет относить все объекты «этого пользователя», и только их. Схема будет удалена из БД также автоматически при удалении пользователя.

Большинство объектов БД, например, таблицы, индексы, процедуры вида SYNONYM, обязаны находиться в одной и ровно одной схеме («принадлежать одному пользователю»). Однако небольшая часть объектов, например, вида PUBLIC SYNONYM или DIRECTORY, хранятся в БД вне всякой схемы и «не принадлежат никакому конкретному пользователю».

Демонстрационный пользователь SCOTT изначально обладает некоторым набором полномочий, например, создания таблиц и подключения к СУБД. В то же время ему изначально не приданы, скажем, полномочия создавать представления данных или же синонимы. Придать ему эти полномочия способны, например, пользователи SYS и SYSTEM. Эти пользователи могут быть названы «администраторами»; они имеются в любой БД Oracle, и при этом SYS может быть неформально назван «суперпользователем» или «главным администратором», а SYSTEM несколько ограничен в правах по сравнению с SYS.

Полномочия (= привилегии) выдаются командой GRANT и изымаются командой REVOKE.

Пример. Подсоединение в SQL*Plus в качестве SYS, простое создание пользователя YARD вместе со своей схемой и с паролем pass, наделение пользователя YARD некоторыми общими привилегиями:

```
CONNECT / AS SYSDBA
CREATE USER yard IDENTIFIED BY pass;
GRANT connect, resource, create view TO yard;
```

Подключение к СУБД под именем YARD и попытка обратиться к таблице EMP:

```
SQL> CONNECT yard/pass
Connected.
SQL> DESCRIBE emp
ERROR:
ORA-04043: object emp does not exist
```

Ошибка возникла потому, что пользователь YARD (как и любой другой) работает в собственной схеме БД, а в схеме YARD таблицы EMP нет — она имеется в схеме SCOTT. Разрешить же пользователю YARD обращаться к данным таблицы EMP в схеме SCOTT можно специально выданной командой GRANT, например:

```
CONNECT scott/tiger  
GRANT SELECT ON emp TO yard;
```

В Oracle понятие «пользователь» естественно связывать не с физическим лицом, обращающимся к БД, а с приложением.

4. Создание, удаление и изменение структуры таблиц

Таблицы представляют из себя главный инструмент моделирования данных в БД, как построенных на основе SQL вообще, так и в Oracle, в частности.

4.1. Предложение CREATE TABLE

Создание таблиц осуществляется предложением CREATE TABLE категории DDL.

Пример:

```
CREATE TABLE proj
(
    projno NUMBER    ( 4 )
  , pname  VARCHAR2 ( 14 )
  , bdate  DATE
  , budget NUMBER    ( 10, 2 )
);
```

Помимо указанных в этом примере, предложение CREATE TABLE может содержать чрезвычайно много прочих конструкций, большая часть которых связана с организацией хранения, а не с логикой данных. Полностью они приводятся в документации Oracle по SQL.

4.2. Типы данных в столбцах

Существующие в Oracle встроенные (предопределенные) типы позволяют указывать столбцам таблицы следующие виды данных.

Встроенные типы

<i>Числа:</i>
типы NUMBER, NUMBER (<i>n</i>), NUMBER (<i>n</i> , <i>m</i>) (в частности NUMBER (*, <i>m</i>)) типы BINARY_FLOAT ^[10-] , BINARY_DOUBLE ^[10-] типы FLOAT, REAL, NUMERIC, DECIMAL, INTEGER и другие, совместимые с ANSI/ISO (и IBM)
<i>Строки текста:</i>
типы VARCHAR2 (<i>n</i>), CHAR (<i>n</i>) типы NVARCHAR2 (<i>n</i>), NCHAR (<i>n</i>) ^[9-] типы CLOB ^[8-] , NCLOB ^[8-] типы STRING, CHARACTER VARYING, NATIONAL CHARACTER VARYING и другие тип LONG ^[уст.]
<i>Строки байтов:</i>
RAW (<i>n</i>) BLOB ^[8-] LONG RAW ^[уст.]
<i>Моменты времени:</i>
DATE TIMESTAMP ^[9,2-] , TIMESTAMP (<i>n</i>) ^[9,2-] TIMESTAMP WITH TIME ZONE ^[9,2-] , TIMESTAMP WITH LOCAL TIME ZONE ^[9,2-]
<i>Интервалы времени:</i>
INTERVAL YEAR TO MONTH ^[9,2-] , INTERVAL DAY TO SECOND ^[9,2-]
<i>Прочие данные:</i>
BFILE ^[8-] ROWID, UROWID ^[8,1-] (физический и «логический физический» адреса строк или объектов в таблицах)
<i>... встроенные объектные:</i>

XMLTYPE ^{[9.2-)} , ANYDATA ^{[9.2-)} , URITYPE с подтипами ^{[9.2-)} , типы для Oracle Spatial ^{[8-)} , для построения геоинформационных систем; прочие

^{[8-)} Начиная с версии 8.

^{[8.1-)} Начиная с версии 8.1.

^{[9-)} Начиная с версии 9.0.

^{[9.2-)} Начиная с версии 9.2.

^{[10-)} Начиная с версии 10.

[уст.] Поддерживается для обратной совместимости.

Пользовательские типы

Объектные ^{[8-)} («структурные»), создаваемые пользователями

^{[8-)} Начиная с версии 8.

Квази-типы

JSON ^{[12-)} (данные в этом формате, но хранение через VARCHAR2, CLOB или BLOB) IDENTITY ^{[12-)} (автоприрастающий тип на основе собственного для столбца породителя номеров; хранение через NUMBER)

^{[12-)} Начиная с версии 12.

Типы BLOB, CLOB, NCLOB и BFILE иногда объединяют в единую категорию типов «больших неструктурированных объектов» (объекты LOB). Для работы с ними в SQL иногда требуется прибегать к функциям встроенного пакета DBMS_LOB, хотя с версии 9.2 поводов для этого стало меньше. Их употребление в SQL связано с определенными ограничениями. Например, столбцы этих типов не могут употребляться в формировании ключа таблицы и вообще индексироваться стандартным образом. Часть подобных ограничений употребления обязаны предположительно гигантским объемам значений, а часть — особенному способу хранения, отличному от принятого для «обычных» данных.

В коде СУБД Oracle сохранились попытки ввести в некоторых случаях более разумные встроенные типы, по ряду причин не доведенные до официального предоставления. Примером может служить тип TIME для обозначения времени суток. Разрешить его употребление в своем сеансе связи с СУБД можно следующей командой (<http://citforum.ru/database/oracle/time/>):

```
ALTER SESSION SET EVENTS '10407 trace name context forever, level 1';
```

Официально этого типа не существует, а следами его фактического наличия являются возможности указывать в обычных выражениях SQL значение времени суток, например, TIME '12:30:45', и некоторые функции, например, TO_TIME и EXTRACT.

Похожим путем уже в версии 8.1 можно было активировать типы TIMESTAMP и INTERVAL (впоследствии ставшие штатными, в отличие от TIME):

```
ALTER SESSION SET EVENTS '10406 trace name context forever';
```

4.2.1. Числовые типы

4.2.1.1. Тип NUMBER

Исторически первым для Oracle числовым типом является NUMBER. Он существует в трех вариантах:

- NUMBER — для хранения чисел «самого общего вида»;
- NUMBER (*n*) — для хранения целых с максимальной точностью мантиссы *n* десятичных позиций;
- NUMBER (*n*, *m*) (в частности NUMBER (*, *m*)) — для хранения чисел «с фиксированной десятичной точкой» с максимальной точностью мантиссы *n* десятичных позиций, из них *m* до десятичной точки.

Формат хранения во всех случаях одинаков:

- 1-й байт — знак числа и степень 100 в двоичном виде;

- следующие байты — двоично-десятичное представление цифр мантииссы, по две десятичные цифры на байт, максимум 38 десятичных цифр
- последний байт — добавочный в случае отрицательных чисел.

Таким образом все варианты типа NUMBER преобразуются к соответствующей им форме при помещении в базу, а при выборке из базы интерпретируются в соответствии с типом столбца. Длина хранения в байтах для NUMBER (p) будет $\text{ROUND} ((\text{LENGTH} (p) + s) / 2) + 1$, где $s = 0$ для положительных чисел и $s = 1$ для отрицательных. Проверить фактическую форму хранения позволяет служебная функция DUMP:

```
SQL> SELECT DUMP ( 1 ) FROM dual;
```

```
DUMP (1)
```

```
-----
Typ=2 Len=2: 193,2
```

```
SQL> SELECT DUMP ( 1000 ) FROM dual;
```

```
DUMP (1000)
```

```
-----
Typ=2 Len=2: 194,11
```

```
SQL> SELECT DUMP ( 1000 - 1 ) FROM dual;
```

```
DUMP (1000-1)
```

```
-----
Typ=2 Len=3: 194,10,100
```

В выдачах Typ = 2 сообщает внутреннее числовое обозначение для типа: NUMBER; Len сообщает общее количество занимаемых байтов, а (десятичные) значения самих байтов перечисляются следом.

4.2.1.2. Подтипы NUMBER

Тип NUMBER в Oracle не входит в стандарт ANSI/ISO SQL. Следующие типы включены в диалект SQL Oracle для совместимости со стандартом и с решениями IBM:

<i>Тип SQL</i>	<i>Совместимость</i>	<i>Соответствующий тип в Oracle</i>
DEC (<i>точность, масштаб</i>)	ANSI/ISO	NUMBER (<i>точность, масштаб</i>)
DEC	ANSI/ISO	NUMBER (38, 0)
DECIMAL (<i>точность, масштаб</i>)	IBM	NUMBER (<i>точность, масштаб</i>)
DECIMAL	IBM	NUMBER (38, 0)
DOUBLE PRECISION	ANSI/ISO	NUMBER
FLOAT (<i>двоичная точность, 1 .. 126</i>)	ANSI/ISO, IBM	NUMBER
INT	ANSI/ISO	NUMBER (38, 0)
INTEGER	ANSI/ISO, IBM	NUMBER (38, 0)
NUMERIC (<i>точность, масштаб</i>)	ANSI/ISO	NUMBER (<i>точность, масштаб</i>)
REAL	ANSI/ISO	NUMBER
SMALLINT	ANSI/ISO, IBM	NUMBER (38, 0)

Содержательно эти типы в Oracle ничего не привносят и фактически являются подтипами типа NUMBER.

4.2.1.3. Типы BINARY_FLOAT и BINARY_DOUBLE

Эти два типа представляют собой второй после NUMBER используемый в Oracle формат хранения чисел, определяемый стандартом IEEE 754 для 32-х и 64-х разрядного внутреннего представления (IEEE Standard for Floating-Point Arithmetic). Стандарт IEEE 754 реализован аппаратно во многих видах процессоров и программно в ряде языков программирования (Java).

Стандарт IEEE 754 определяет больше, чем просто формат хранения чисел. Он, например, предусматривает особые значения «не число» (Not a Number) и +/- бесконечность. В Oracle точно воспроизведен формат хранения, но не все прочие подробности — стандарта IEEE 754. Возможность сослаться на особые «значения» дают следующие обозначения:

```
BINARY_FLOAT_NAN
BINARY_FLOAT_INFINITY
BINARY_DOUBLE_NAN
BINARY_DOUBLE_INFINITY
```

Например:

```
SQL> SELECT -BINARY_FLOAT_INFINITY FROM dual;
```

```
-BINARY_FLOAT_INFINITY
-----
-Inf
```

Из-за несовместимости форматов простая передача данных из вида NUMBER в BINARY_FLOAT/DOUBLE и обратно может приводить к потере точности. Зато при общении СУБД посредством этих двух форматов с внешними средами, работающими на основе стандарта IEEE 754, точность, наоборот, будет сохраняться.

4.2.2. Строки текста

«Обычные» строки («короткие» и в основной кодировке БД) хранятся в полях типов VARCHAR2 (*n*) и CHAR (*n*). *n* задает для VARCHAR2 максимально допустимое число хранимых символов (длиною до 4000 байтов с версии 8 и до 2000 байтов ранее), а для CHAR — всегда одно и то же, фиксированное (до 2000 байтов). С версии 12 максимальная длина доведена до 32767 байт (при условии, что параметр СУБД MAX_STRING_SIZE = EXTENDED).

«Короткие» строки в дополнительной (так называемой «национальной») кодировке БД, которая всегда многобайтовая, хранятся в полях типов NVARCHAR2 (*n*) и NCHAR (*n*).

Для многобайтовой кодировки длину можно указывать как в символах (*n* CHAR), так и в байтах (*n* BYTE); по умолчанию считается последнее. Синтаксически равносильно допускается написание CHAR *n* и BYTE *n*. При создании БД основной кодировкой может быть объявлена многобайтовая (пока это делается нечасто, но распространенность растет), поэтому определения вида VARCHAR2 (5 CHAR) или CHAR (10 BYTE) также приемлемы. Синтаксически допускается проставить ключевые слова CHAR и BYTE после числа, например, VARCHAR2 (CHAR 5) и CHAR (BYTE 10).

Типы CLOB и NCLOB (character LOB — large object — и national character LOB) используются для хранения «больших» строк текста длиною до 4 Гб - 1 до версии 9 включительно, и до нескольких Тб в версиях 10+. Сами значения этих типов обычно хранятся в БД отдельно от обычных данных таблицы, и доступ к ним осуществляется посредством так называемых «локаторов», однако для программирования на SQL эти подробности часто незаметны.

Тип VARCHAR2 не определен стандартом SQL и отличается от типа VARCHAR последнего тем, что полагает строку без символов отсутствующей строкой. Тип CLOB воплощен в Oracle в соответствии со стандартом.

Типы CHAR и NCHAR выделяются из всех прочих тем, что способны исказить присваиваемую величину. Если присвоить полям этих типов величину, более короткую, чем для них объявлено, в БД появится значение, дополненное справа пробелами. Так, если столбец объявлен как CHAR (10), то при добавлении в него символьных строк ' ', ' ', ' ', и так далее, в БД будет занесена строка из 10 пробелов.

Типы STRING, CHARACTER VARYING, NATIONAL CHARACTER VARYING и другие, совместимые со стандартом SQL ANSI/ISO, сводятся в Oracle SQL к VARCHAR2 и NVARCHAR2 и, таким образом, вторичны.

4.2.3. Строки байтов

Тип RAW (*n*) аналогичен VARCHAR2 (*n*) с разницей максимально разрешенной длины — вплоть до 2000 байтов в версиях до 12, и 32767 байтов с версии 12.

Тип BLOB (binary LOB) аналогичен CLOB и по сути описывает файл (поток байтов), возможно, очень большой и размещаемый в БД (речь при этом *не* идет о воспроизведении в БД Oracle файловой системы). С версии 11 такое помещение файла в БД Oracle может (в силу обновления механизма внутренней организации) сопровождаться сокращением объемов хранения и оказаться выгодным с точки зрения компактности.

Тип BFILE (binary file) аналогичен BLOB, но только сам поток байтов хранится вне БД, а именно в файле. Утилитарно можно полагать его типом ссылки на файл ОС, где работает СУБД, то есть «на сервере». Тип может показаться удобным, так как облегчает доступ к данным, достижимым из Oracle, со стороны посторонних программ; однако он имеет ту особенность, что сами данные, расположенные вне БД, не затрагиваются процедурами резервного копирования и восстановления, а также командами управлением транзакциями. В БД хранится только ссылка на внешний файл.

LONG (для строк текста) и LONG RAW (для строк из байтов) — устаревшие типы, сохраняемые ради обратной совместимости. Например, они встречаются в некоторых системных таблицах, спроектированных в прежние времена.

4.2.4. Моменты и интервалы времени

Тип DATE в Oracle не совпадает с одноименным типом в стандарте SQL и рассчитан на хранение одновременно шести компонент момента времени: года, месяца, числа, часа, минут и секунд. По сути он имеет в Oracle собственную, нестандартную реализацию. Шестикомпонентность типа DATE доставляет неудобства, когда требуется сохранить в БД только дату или только время суток.

Типы TIMESTAMP стандартны. В основном варианте TIMESTAMP хранит те же шесть компонент, что и DATE, но с точностью до наносекунд. Хранимую точность можно намеренно ограничить, указав в скобках *n* от 0 до 9. В расширенных вариантах тип включает еще зону времени (часовой пояс).

Интервальные типы INTERVAL YEAR TO MONTH и INTERVAL DAY TO SECOND соответствуют стандарту и позволяют хранить «грубые» интервалы (исчисляемые годами и месяцами) и «точные» (исчисляемые днями, часами, минутами и секундами).

4.3. Общие свойства типов

У типов данных в Oracle имеются общие свойства:

- у всех типов дополнительно ко множеству допустимых величин наличествует особое «значение», обозначаемое в выражениях как NULL;
- большинство типов допускает сравнение на равенство.

4.3.1. Неуказанные значения и NULL

Несуществующие значения, явно обозначаемые (при необходимости) как NULL, можно воспринимать формально (и механически следовать правилам выполнения операций с NULL), но интерпретировать их указание в столбце таблицы можно пытаться по-разному: как то, что «значение отсутствует» (например,

«сотрудник не получал комиссионных») и то, что значение «неизвестно какое» из допустимых (например, «сотрудник получил какие-то комиссионные, но БД неизвестно, какие именно»). К сожалению это содержательно не одно и то же, равно как и наделения NULL этими двумя смыслами при моделировании не достаточно (хотя и хватает для описания 99% жизненных ситуаций). По этим причинам, хотя на первый взгляд возможность опустить значение в столбце за его отсутствием может показаться и привлекательной, последующая работа с такими данными доставит программисту значительно больше неприятностей из-за необходимости учета при составлении запросов не формализованных в БД сведений о данных, из-за усложнения запросов и из-за неизбежных рисков ошибиться при программировании. Примеры подобных неприятностей обозначены в разных местах текста ниже.

Стандарт SQL допускает в таблицах NULL, но оговаривает, что во имя надежности обращения к данным программисту следует всячески избегать пропусков значений в столбцах либо уж употреблять их лишь в смысле «значение неизвестно» (unknown). Стандарт называет NULL особым *значением*, общим для всех типов, и обрекает себя тем самым на критику со стороны специалистов, полагающих, что правильнее назвать NULL *символом*⁴, так как он не удовлетворяет признакам значения. Действительно, если x «имеет значение» NULL, то $x = x$ не истина, что довольно необычно. Это уже проблема не интерпретации, а принятых правил употребления.

Oracle наследует все проблемы с NULL стандартного SQL, но, как показывает опыт, рассматривает NULL принадлежностью каждого типа в отдельности, что иногда дает о себе знать в попытках сформулировать некорректное с точки зрения Oracle выражение.

Что касается реляционной теории, то первые 10 лет своего существования она не рассматривала пропущенных значений вовсе, и не имела дела с NULL. Позже мнения специалистов разошлись: одни (в их числе Э. Кодд) посчитали возможным (хотя и «нехотая», нежелательным на деле) использование пропущенных значений, а другие настаивали на их недопущении. То есть разрешение значениям в базе отсутствовать, вместе с вытекающей из этого необходимостью громоздкого аппарата их учета, можно считать одним из предлагаемых расширений, притом, спорным, реляционной модели.

Крайнюю позицию занимает К. Дейт. По его мнению пропущенным значениям в реляционной модели не место, а все задачи, которые предлагают решать с их помощью, можно решить, не прибегая к ним.

Например, если требуется учесть возможность отсутствия комиссионных у работника, атрибут «комиссионные» можно исключить из отношения «сотрудники» и перенести в дополнительное отношение:

⁴ Смотри книгу Дейт К. Дж., Дарвен Х. Основы будущих систем баз данных. Третий манифест. Перевод с английского. Изд. 2, 2004. Примечательно, что один из критиков – автор этой книги, в свое время входил в состав одного из национальных комитетов ISO по стандартизации SQL; это красноречиво характеризует демократические институты, к числу которых относится ISO.

Сотрудники

7369	Smith	800		...
7499	Allen	1600	300	...
7521	Ward	1250	500	...



Сотрудники

7369	Smith	800	...
7499	Allen	1600	...
7521	Ward	1250	...

Комиссионные

7499	300
7521	500

внешний ключ

То же в SQL можно сделать для таблиц, но там, к сожалению, это усложнит запрос к данным, а в реальных системах еще и замедлит вычисление.

В некоторых случаях удастся обойтись и более простым решением. Например, если для хранения почтового индекса используется столбец типа CHAR (6), то отсутствие индекса можно обозначить пробелами. Недостаток в том, что обработка отсутствующих значений перестанет быть тогда универсальной.

4.3.2. Сравнение величин на равенство

Сравнение величин на равенство — более безобидное и очевидное свойство типов в Oracle. Однако для некоторых встроенных «сложных» типов, не говоря уже об объектных, созданных программистом, оно не выполняется. Например, Oracle не позволяет сравнить в запросе SQL две величины типов LOB или XMLTYPE. В приводимом ниже доказательстве команда VARIABLE предназначена для определения в SQL*Plus переменной указанного типа и не имеет отношения к SQL:

```
VARIABLE x CLOB
VARIABLE y CLOB
```

```
SELECT 'ok' FROM dual WHERE :x = :y;
-- ошибка !
```

Другая невоодушевляющая особенность сравнения на равенство лежит в области формулировки действия и связана как раз с отсутствующими значениями. Она упоминалась только что. Сравнения с NULL требуют самостоятельного оформления, пример коего последует позже.

4.4. Уточнения возможных значений в столбцах

Кроме необходимого в описании столбца типа, можно сообщить дополнительные правила для допустимых значений в полях добавляемых или обновляемых строк таблицы.

Пример:

```
CREATE TABLE projx
(
    projno NUMBER      (4) NOT NULL CHECK ( projno > 0)
```

```
, pname VARCHAR2 (14) CHECK (SUBSTR(pname,1,1) BETWEEN 'A' AND 'Z')
, bdate DATE DEFAULT TRUNC ( SYSDATE )
, budget NUMBER (10,2) CHECK ( budget >= 0)
);
```

Такие правила делятся на две категории: значения данных по умолчанию и «ограничения» (или же «правила») «целостности».

Приписка **DEFAULT** *выражение* к определению столбца указывает на значение, которое будет заноситься СУБД в поле добавляемой строки, если программист в операции **INSERT** никакого значения для этого поля не привел.

Упражнение. Проверить результаты изменений данных:

```
INSERT INTO projx ( projno, bdate ) VALUES ( 15, SYSDATE + 1 );
INSERT INTO projx ( projno ) VALUES ( 16 );

SELECT * FROM projx;

UPDATE projx SET bdate = DATE '2009-09-17' WHERE projno = 15;

SELECT * FROM projx;
```

В выражении после слова **DEFAULT** нельзя обращаться к данным в БД и к величинам, зависящим от конкретной строки, а в остальном сгодится любая суперпозиция функций SQL. В примере выше выражение построено применением функции **TRUNC** к системной переменной **SYSDATE**. Значение для **SYSDATE** СУБД берет с часов компьютера, на котором работает.

К «ограничениям целостности» в примере выше относятся уточнения описаний столбцов, оформленные с помощью слов **NOT NULL** и **CHECK**. Более систематично они вместе с другими разрешенными ограничениями целостности будут описаны в соответствующем разделе ниже.

Особый вариант **DEFAULT ON NULL** *выражение* (в Oracle 12+) есть сочетание указания значения по умолчанию при **INSERT** и ограничения целостности (**NOT NULL**); введен для удобства.

4.5. Свойства столбцов, не связанные со значениями

4.5.1. Шифрованное хранение

С версии 10.2 Enterprise Edition, (а) при установленном дополнении к СУБД Advanced Security Option и (б) при наличии предварительно созданного на сервере «бумажника» Oracle Wallet можно потребовать автоматического («прозрачного») шифрования значений столбца при помещении их в БД и автоматической дешифровки при извлечении. Пример указания трех разных технических способов шифрования для трех разных столбцов:

```
...
, sal      NUMBER ( 7, 2 ) ENCRYPT
, comm     NUMBER ( 7, 2 ) ENCRYPT NO SALT
, hiredate DATE ENCRYPT USING 'AES256' IDENTIFIED BY 'SecretWord'
...
```

Свойство «хранить в зашифрованном виде» — нефиксированное: его можно добавлять или отменять для столбцов существующей таблицы при том, что эти действия будут сопровождаться автоматической правкой ранее введенных в таблице данных.

Прозрачному шифрованию подлежат только основные встроенные типы (с версии 11 плюс LOB, хотя и отдельным способом), и на употребление этой возможности наложены некоторые ограничения.

4.5.2. Виртуальные столбцы

Версия 11 открыла возможность объявлять в таблице виртуальные (мнимые, воображаемые) столбцы (для внутреннего употребления они существовали в Oracle с версии 8, и служили инструментом реализации объектных возможностей). Значения в них не хранятся в БД самостоятельно, а вычисляются автоматически при запрашивании на основе действительных значений других полей строки (тем самым они не нарушают 3-ю нормальную форму). Например, в таблице EMP могло бы иметься такое определение:

```
...
, sal      NUMBER ( 7, 2 )
, comm     NUMBER ( 7, 2 )
, earnings AS ( sal + comm )
...
```

Дополнительные ключевые слова помогут при этом сделать запись яснее, не меняя сути формулировки, например:

```
... earnings AS GENERATED ALWAYS ( sal + comm ) ...
```

Виртуальные столбцы, подобно действительным, можно индексировать (индекс при этом получается с «функционально преобразованным значением ключа») и использовать в формулировании ограничений целостности.

4.5.3. Невидимые столбцы

Версия 12 открыла возможность объявлять в таблице невидимые столбцы (для внутреннего употребления они существовали в Oracle с версии 8, и служили инструментом реализации объектных возможностей), например:

```
...
, sal      NUMBER ( 7, 2 )
, comm     NUMBER ( 7, 2 ) INVISIBLE
...
```

В обычных операциях такие столбцы не видны, но явным указанием имени можно к ним обращаться.

Перевод столбца в состояние INVISIBLE помогает планировать отказ от его использования.

4.6. Создание таблиц по результатам запроса к БД

Второй способ завести таблицы в SQL состоит не в явном перечислении столбцов и их свойств, а ссылке на результат запроса к БД:

```
CREATE TABLE dept_copy AS SELECT * FROM dept;
```

Запрос следует после ключевого слова AS и выше указан чрезвычайно простым, но имеет право быть и сколь угодно сложным. Сначала он вычисляется СУБД, после чего в БД создается таблица со структурой полученного результата (с воспроизведением всех столбцов с их типами), и эта новая таблица тут же заполняется данными результата. Такой способ создания таблицы позволяет экономить усилия программиста и время, когда новую таблицу нужно создать на основе уже имеющихся данных.

Следующий пример показывает, как в таких случаях можно распоряжаться именами столбцов в новой таблице, не копируя слепо имена столбцов из результата запроса:

```
CREATE TABLE emps ( name, department )
AS
```

```

SELECT INITCAP ( ename ), INITCAP ( dname )
FROM   emp, dept
WHERE  emp.deptno = dept.deptno
;

```

Иногда, как возможно в первом примере, команду CREATE TABLE ... AS используют для «копирования» существующей таблицы, указав запрос в форме SELECT * FROM *таблица*. В таких случаях не следует забывать, что в новой таблице из структурных свойств старой окажутся воспроизведены только типы столбцов и свойства NULL/NOT NULL, но не ограничения целостности и не DEFAULT. При этом воспроизведение свойств NULL/NOT NULL столбцов довольно необычно, так как оно не имеет смысла для запросов с более общей формулировкой.

В случае запроса общего вида, ограничения целостности можно указывать сразу при создании таблицы:

```

CREATE TABLE emps1 ( name PRIMARY KEY, department NOT NULL )
AS
  SELECT INITCAP ( ename ), INITCAP ( dname )
 FROM   emp, dept
WHERE  emp.deptno = dept.deptno
;

```

Логически подобные указания необязательны, так как можно создать таблицу без ограничений целостности, и тут же следом их добавить. Однако технологически такое создание таблицы одной командой SQL будет происходить быстрее.

4.7. Именование таблиц и столбцов

Правила именования таблиц и столбцов

- Две таблицы одной схемы (или принадлежащие одному владельцу, что в Oracle одно и то же) должны иметь разные имена.
- Два столбца одной таблицы должны иметь разные имена.
- Длина имени может достигать не более 30-и байтов (в версии 11 словарь-справочник стал допускать имена длиной до 128-и байтов, однако для обычных объектов внутренняя программная логика ориентируется на прежнее ограничение). Типично это соответствует 30-и знакам, однако когда в качестве основной кодировки БД выбрана Unicode, то нелатинские буквы требуют более одного байта, и допустимый предел длины имени может оказаться менее 30-и знаков.
- «Обычное» имя может состоять только из букв, цифр и символов `_`, `$`, `#` и начинаться с буквы. Будучи заключены в двойные кавычки, имена могут состоять из любых символов.
- Имена не должны совпадать с зарезервированными в Oracle словами.
- Русские названия допускаются без ограничения употребления, если только для БД указана одна из «русских» кодировок.

Имя объекта, указанное в предложении SQL без обрамляющих двойных кавычек, сопоставляется с именами из словаря-справочника БД, будучи приведенным к верхнему регистру. Двойные кавычки предотвращают повышение регистра.

Упражнение. Выполните последовательно и сравните ответы СУБД:

```

CREATE TABLE t ( a NUMBER, a VARCHAR2 ( 1 ) );

CREATE TABLE t ( a NUMBER, "a" VARCHAR2 ( 1 ) );

CREATE TABLE t ( a NUMBER, "a" VARCHAR2 ( 1 ) );

CREATE TABLE "t" ( a NUMBER, "a" VARCHAR2 ( 1 ) );

```

Замечания.

- (1) Заключение имени в двойные кавычки обычно мера вынужденная, так как влечет обязательность указания двойных кавычек и в дальнейшем, после создания таблицы (иначе СУБД все время будет пытаться повысить регистр символов), то есть неудобства употребления.
- (2) Понятие «зарезервированное слово» в силу разных причин определено в Oracle нечетко. В Oracle SQL и в PL/SQL множества зарезервированных слов, большей частью совпадая, все же различаются. Например, слово `TIMESTAMP` *можно* использовать для именования столбца.
- (3) Использование русских букв в именах не влечет никаких неприятностей со стороны СУБД. Тем не менее внешние по отношению к СУБД программы не всегда умеют их правильно обрабатывать. В силу этого к русским именам таблиц и столбцов в Oracle следует относиться настороженно.

Oracle допускает в SQL ссылки не только на односоставные имена таблиц, но и на составные.

Так, имя таблицы в команде SQL может быть уточнено именем схемы, в которой числится эта таблица, например: `SCOTT.DEPT`, `SYS.OBJ$`. В действительности, если в запросе приведено односоставное имя, то при обработке Oracle самостоятельно дополнит его именем схемы. Обычно это будет имя схемы, с которой работает программа (что в Oracle равнозначно имени пользователя), но при желании такое подразумеваемое расширение имени таблицы можно заменить в пределах отдельного сеанса на имя любой другой *существующей* в данный момент в БД схемы, например:

```
ALTER SESSION SET CURRENT_SCHEMA = yard;
```

После этого обращения просто к EMP будут считаться обращениями к `YARD.EMP`, а не к `SCOTT.EMP`, как по умолчанию. Этим иногда пользуются при разработке приложения для придания ему гибкости. Подобная подмена умолчательного имени схемы не влечет несанкционированного доступа к объектам чужой схемы, так как будет означать только *попытку* обращения к чужому объекту. Окажется ли успешным обращение, определяется совсем другим механизмом полномочий доступа (прав).

Дополнительно в обращении к таблице можно указать имя заведенной предварительно ссылки на другую БД, с которой установлена связь, и тогда имя может в конечном итоге оказаться трехсоставным, например: `SCOTT.EMP@PERSONNELDB`. Этим обозначено обращение к таблице EMP в схеме SCOTT БД, именованной как PERSONNELDB.

4.8. Удаление таблиц

Удаление таблицы выполняется командой `DROP TABLE имя_таблицы`.

Сведения о таблице СУБД хранит в двух отдельных местах БД: в справочной части БД (в словаре-справочнике) — описание и в виде самостоятельной структуры (сегмента) в файлах табличного пространства — содержимое, данные.

До версии 10 единственным способом удаления по команде `DROP TABLE` было удаление из словаря-справочника сведений о таблице и удаление сегмента с данными из «рабочей части» БД. С версии 10 возможен (и действует автоматически по умолчанию) второй вариант, когда по команде `DROP TABLE` описание таблицы и сегмент с ее данными получают новое системное имя, после чего таблица под прежним именем перестает существовать, но фактически какое-то время может быть еще доступна, и даже восстановлена. Этот второй способ удаления таблицы воплощает идею «мусорной корзины».

4.8.1. Простое удаление

Пример простой команды удаления таблицы:

```
DROP TABLE dept_copy2;
```

Если на столбцы таблицы определены ссылки внешними ключами других таблиц, СУБД не позволит выполнить `DROP TABLE` и вернет ошибку. (Замечания: (а) это связано не с наличием конкретных

ссылающихся строк, а с наличием самого правила внешнего ключа; (б) внешний ключ, ссылающийся на значения столбцов «собственной» таблицы, не препятствует ее удалению). Конструкция CASCADE CONSTRAINTS в команде DROP TABLE позволит-таки удалить таблицу, но при этом СУБД удалит сначала «мешающее» правило внешнего ключа. Столбцы другой, оставшейся таблицы в результате сохраняют свои значения, но они уже не будут обременены ограничением ссылочной целостности. Фактически использование CASCADE CONSTRAINTS равносильно последовательному удалению всех правил внешнего ключа, имеющих адресатом таблицу (таковых может быть несколько), и выполнению в завершение простой команды DROP TABLE.

Например, если на столбцы таблицы DEPT_COPY определены ссылки внешними ключами из других таблиц, настоять на удалении DEPT_COPY можно, выдав:

```
DROP TABLE dept_copy2 CASCADE CONSTRAINTS;
```

4.8.2. Мусорная корзина

С версии 10 смысл команды DROP изменился. В основном случае после нее и описание, и данные таблицы продолжают храниться на своих местах, но под новыми, присвоенными системой автоматически, именами. Для пользователя таблица, как и прежде, пропала, однако на деле все, что нужно для ее восстановления, буде такая необходимость возникнет, продолжает храниться в БД. Тем самым для таблиц реализована техника мусорной корзины (recycle bin), хорошо известная по файловым системам.

Список содержимого мусорной корзины можно получить из системной таблицы USER_RECYCLEBIN (публичный синоним — RECYCLEBIN):

```
SELECT object_name, original_name, droptime FROM user_recyclebin;
```

Восстановить таблицу по *исходному имени* (столбец ORIGINAL_NAME из USER_RECYCLEBIN) можно, например, так:

```
FLASHBACK TABLE dept_copy2 TO BEFORE DROP;
```

Восстанавливается не все и не точно. У восстановленной подобным образом таблицы будут отсутствовать, возможно, имевшиеся до удаления ограничения целостности, а индексы, хотя и восстановятся, но с системными именами.

Для удаления из мусорной корзины нужно использовать команду PURGE, например:

```
PURGE TABLE dept_copy3;
```

```
PURGE RECYCLEBIN;
```

Чтобы таблица удалялись сразу безвозвратно, следует использовать конструкцию PURGE в команде DROP, например:

```
DROP TABLE dept_copy3 PURGE;
```

Умолчательное помещение таблицы в мусорную корзину можно отменить и вернуться к старой обработке команды DROP. Для этого нужно задать значение OFF параметру СУБД RECYCLEBIN. Последний допускает динамическую установку на уровне СУБД (ALTER SYSTEM ...) и на уровне сеанса (ALTER SESSION ...), например:

```
ALTER SESSION SET RECYCLEBIN = OFF;
```

Как и в файловых системах, в Oracle мусорная корзина хранит содержимое до поры до времени. В основном она — инструмент восстановления данных после ошибочных действий пользователя, возникших в результате проведения опытов с базой или по неосторожности.

4.9. Изменение структуры таблиц

Изменение структуры и других свойств существующей таблицы необязательно требует удаления ее и воссоздания в переделаном виде. Часто оно осуществимо «по живому», что менее хлопотно для программиста и быстрее. Oracle не чинит чрезмерных препятствий таким изменениям и, как правило, допустит их, если они:

- не вступают в логические противоречия с описанием таблицы в БД, и
- не противоречат уже занесенным в таблицу данным.

В общем допустимы следующие структурные изменения: добавление и убирание столбцов, изменение типа столбца, добавление и убирание ограничений целостности данных. Все они осуществляются разновидностями команды ALTER TABLE.

С версии 11.2 Oracle предлагает к тому же особую технику редакций объектов хранения для внесения изменений в схему данных. Эта техника не рассматривается непосредственно здесь, но в одном из разделов ниже.

Кроме того, с версии 9 Oracle дает возможность переопределять структуру существующих таблиц не средствами SQL (в принципе достаточными для этой цели), а программно с помощью встроенного системного пакета DBMS_REDEFINITION. Делается это исключительно по технологическим соображениям с тем, чтобы время недоступности данных при перестройке было по возможности малым (заявлено, что переопределение происходит online, то есть без прекращения доступности вовсе). Главным образом это актуально для таблиц с большими объемами данных при существующих жестких требованиях к доступности. Такая техника в настоящем тексте, посвященном SQL, естественно, не затрагивается.

4.9.1. Добавление столбца

Добавление столбца не связано с какими бы то ни было логическими противоречиями и осуществимо почти всегда. Пример:

```
ALTER TABLE projx ADD ( pcode VARCHAR2 ( 1 ) );
```

Единственным логическим препятствием к добавлению столбца служит достижение предельного количества столбцов в таблице Oracle — 1000 (такое значение дает документация по Oracle).

4.9.2. Изменение типа столбца

Выполняется с использованием слова MODIFY, например:

```
ALTER TABLE projx MODIFY ( pcode NUMBER ( 6 ) );
```

Логическим препятствием к выполнению такого действия может оказаться вступление в противоречие с заданным ранее ограничением (правилом) целостности, в определении которого участвует столбец. Например наличие внешнего ключа требует согласованности типов ссылающихся столбцов и столбцов-адресатов. Так, сослаться столбцом типа NUMBER на столбец типа VARCHAR2 нельзя.

Техническим препятствием к изменению типа может служить необходимость переформатировать данные в столбце, что здесь не допускается. Как следствие:

- тип столбца можно поменять произвольно (TIMESTAMP на VARCHAR2 и так далее), когда столбец целиком пуст, или строки в таблице отсутствуют;
- если данные в столбце есть, возможна замена только:

- в пределах однородных типов (с одинаковым форматом хранения), например всех разновидностей NUMBER друг на друга, VARCHAR2 на CHAR и обратно;
- и если это допускают фактические данные: (а) всегда можно увеличить точность хранения в столбце, но (б) уменьшить точность хранения при наличии данных нельзя.

Тип DATE хранится как TIMESTAMP (0) и однороден с ним в указанном смысле, а вот TIMESTAMP WITH TIME ZONE не однороден не только с DATE, но и TIMESTAMP, и допускает взаимные замены с ними только на пустом столбце.

Особый случай дают типы LOB, величины которых хранятся непохоже от всех остальных. Столбцу типов LOB нельзя назначить другой тип, и другие типы невозможно заменить на LOB, даже на пустом столбце. При необходимости такой столбец придется удалить и воссоздать с другим типом.

4.9.3. Добавление и упразднение умолчательных значений в столбце и ограничений целостности

Выполняется с помощью ключевых слов ADD и DROP или же MODIFY (с отчасти различной областью применимости).

Пример употребления слова MODIFY для добавления и для снятия умолчательного значения в поле добавляемых строк:

```
ALTER TABLE emp MODIFY ( ename DEFAULT UPPER ( USER ) );
```

```
ALTER TABLE emp MODIFY (ename DEFAULT NULL );
```

Пример употребления слов ADD и DROP с целью добавления и снятия ограничения первичного ключа:

```
ALTER TABLE projx ADD PRIMARY KEY ( projno );
```

```
ALTER TABLE projx DROP PRIMARY KEY;
```

Другие примеры и пояснения последуют далее в самостоятельном разделе.

Со стороны данных препятствий к упразднению ранее существовавших ограничений целостности нет. Препятствие может возникнуть со стороны описания данных, когда имеется зависимое другое ограничение целостности, как это происходит в случае внешнего ключа (тогда нельзя упразднить уникальность или же правило первичного ключа, на которые ссылается какой-нибудь внешний ключ).

Препятствием к добавлению ограничения целостности может служить вступление в противоречие с имеющимися в таблице данными. Например, если в столбце обнаруживается отсутствие значения (хотя бы в поле одной строки), добавить к определению столбца правило NOT NULL не удастся; наличие повторяющихся значений в столбцах не позволит завести правило уникальности или первичного ключа, и так далее.

В некоторых случаях такое противоречие со сложившимися в таблице данными можно преодолевать, о чем будет сказано ниже.

4.9.4. Удаление столбца

Возможно с версии Oracle 8. Примеры:

```
ALTER TABLE projx DROP ( pcode );
```

```
ALTER TABLE projx DROP COLUMN pcode;
```

Здесь указаны две формы выполнения одного и того же, разве что первая форма носит более общий характер и рассчитана на возможность разом удалить группу столбцов.

Логическим препятствием для удаления столбца может оказаться его участие в определении внешнего ключа в качестве адресата. Не будь этого, по удалению столбца в БД осталось бы некорректно определенное правило внешнего ключа. Вместо последовательного упразднения мешающего ограничения и удаления столбца программист может обойтись одной командой, что быстрее и короче:

```
ALTER TABLE projx DROP ( pcode ) CASCADE CONSTRAINTS;
```

Значения в столбцах бывшего внешнего ключа, остаются, но уже не обремененные ссылочной целостностью.

4.9.5. Технологии повышения производительности

Выполнение некоторых изменений в описании таблицы требует правки самих данных, и когда данных много, оно сопряжено с большими затратами. Ниже приводятся два примера технологических ухищрений, предлагаемых Oracle для снижения подобных затрат.

4.9.5.1. Добавление столбца

Это замечание касается не виртуальных (с версии 11), а реальных добавляемых столбцов.

В типичном случае такой добавляемый столбец не будет иметь значений, и его добавление не потребует правки в БД существующих строк таблицы, то есть совершится со скоростью внесения изменений в описание таблицы в словаре-справочнике. Однако, если для нового столбца указано свойство DEFAULT, то системе потребуется внести значение в новое поле у всех имеющихся строк. Если строк много, на это уходит заметное время. С версии 11 действует оптимизация такого исправления данных для случаев, когда вместе с DEFAULT для столбца одновременно указано NOT NULL. В самом деле, наличие этих свойств обоих сразу позволяет отказаться от фактического добавления значения в каждую строку таблицы, а вместо этого единожды сохранить значение выражения, указанного во фразе DEFAULT, в описании таблицы, а при последующих запросах к строке имитировать наличие этого значения в добавленном поле. С версии 11 СУБД Oracle так и поступает, экономя, вдобавок, дисковое пространство.

Такое поведение СУБД не оформлено на уровне SQL и действует автоматически, в отличие от следующего ниже.

4.9.5.2. Удаление столбца

Следующие замечания не касаются удаления виртуальных столбцов (в версиях 11 и выше), но только реальных. Удаление реального столбца влечет физическое переформатирование всех строк таблицы и на больших объемах данных сопряжено с большими затратами. В качестве выхода из положения предлагается два решения.

Следующее указание позволит ускорить удаление столбца за счет выполнения контрольной точки автоматически после правки каждых 1000 строк (и тогда сегмент отмены не будет расти чрезмерно):

```
ALTER TABLE projx DROP ( pcode ) CHECKPOINT 1000;
```

Если в процессе удаления произошел сбой, процедуру можно продолжить командой:

```
ALTER TABLE projx DROP COLUMNS CONTINUE;
```

Ее также можно продолжить указанием CHECKPOINT *nnnn*:

```
ALTER TABLE projx DROP COLUMNS CONTINUE CHECKPOINT 1000;
```

Другой способ — объявить сначала столбец «неиспользуемым», что не приведет к физическому перебору имеющихся записей:

```
ALTER TABLE projx SET UNUSED COLUMN pcode CASCADE CONSTRAINTS;
```

Для программы таблица мгновенно окажется без столбца, хотя в базе данные столбца никуда не денутся. СУБД сама их будет изымать из полей по мере обращения к строкам по текущим поводам, то есть заодно. В конечном итоге фактическое переформатирование строк таблицы может растянуться на очень долгое время. Однако при желании его можно будет волевым порядком завершить. Для этого нужно дождаться невысокой загрузки СУБД обычной работой и выдать по образцу следующего:

```
ALTER TABLE projx DROP UNUSED COLUMNS;
```

Такую команду опять-таки можно продолжить оптимизирующим уточнением CHECKPOINT *nnnn*.

4.10. Переименования

Неудачно или несвоевременно названную таблицу можно переименовать, например:

```
ALTER TABLE emp RENAME TO employee;
```

Другой способ:

```
RENAME employee TO emp;
```

Команда RENAME по сравнению с более сфокусированной ALTER TABLE ... RENAME является более общей, так как позволяет переименовать помимо таблиц объекты некоторых других типов: представление данных (view), генератор последовательности (sequence) и частный синоним. Ее название унаследовано языком SQL от реляционной модели, где имеется одноименная операция.

Пример переименования столбца:

```
ALTER TABLE projx RENAME COLUMN pcode TO project_code;
```

При переименовании объекта СУБД пометит свойства зависимых от данного объектов (например, хранимых процедур на PL/SQL, обращающихся к данной таблице) признаком INVALID, что вызовет потребность их перекомпиляции перед очередным обращением к ним. Ссылки же на таблицу из внешних программ находятся вне компетенции БД; изменение имен пройдет для таких программ незаметно, но в то же время они потеряют свою работоспособность. Это обстоятельство существенно уменьшает ценность операции переименования в реальной практике.

Переименовать таблицу можно и непосредственной правкой таблицы OBJ\$ словаря-справочника (см. далее). Такой же подход позволяет переименовать и столбцы таблиц, путем внесения правки в таблицу COL\$. Однако применять его следует только опытным пользователям и с осторожностью. Он осуществим только для пользователей, имеющих доступ к этим таблицам схемы SYS, и к тому же не изменит состояния зависимых от таблицы подпрограмм на значение INVALID.

4.11. Использование синонимов для именования таблиц

Синонимы позволяют завести *дополнительные* имена для обращения к таблице, не отменяя основного имени:

```
CREATE SYNONYM members FOR emp;
```

```
SELECT * FROM emp;
```

```
SELECT * FROM members;
```

Теперь, обнаружив обращение к MEMBERS, СУБД определит по своей справочной информации, что это синоним имени EMP, и обратится фактически к EMP. Возможность прямого обращения по имени EMP в тексте команды SQL при этом не теряется, и на работе старых программ появление у таблицы синонимов никак не скажется. Это, однако, не касается команд DDL ALTER/DROP TABLE, где ссылаться следует только на истинное имя таблицы (в полном соответствии с синтаксисом, так как в этих командах используется именно ключевое слово TABLE).

На практике синонимы заводятся с разными целями:

- присвоить таблице имя, больше подходящее ее содержанию;
- упростить имя таблицы в запросе, например, OBJECTS вместо SYS.OBJ\$;
- замаскировать обращение к таблице в другой БД или в другой схеме для придания гибкости кода.

Удаление выполняется командой DROP SYNONYM:

```
DROP SYNONYM members;
```

Синоним, заведенный в схеме (например SCOTT), как и таблица, сам становится объектом схемы, доступным изначально только пользователю-хозяину схемы или же администратору с соответствующим полномочием. Однако Oracle позволяет создавать еще и PUBLIC SYNONYM: «внесхемный», общедоступный синоним. Публичные синонимы активно используются в административной части БД Oracle, но нередко и обычными разработчиками в своих целях. В силу того, что пространства имен публичных и схемных синонимов разные, возможны «спорные» ситуации:

```
CREATE PUBLIC SYNONYM syn1 FOR dept;
-- публичный синоним
CREATE SYNONYM syn1 FOR emp;
-- собственный синоним схемы
SELECT * FROM syn1;
-- DEPT или EMP ?
```

По существующему правилу разрешения имен Oracle отдаст в последнем запросе предпочтение схемному («частному») синониму. Получается, что появление частного синонима в некоторых случаях способно изменять смысл существовавшего в программе до этого запроса, о чем не следует забывать разработчику (осознанно это можно использовать в своих целях, а по незнанию или забывчивости можно обрести неприятности).

Создание синонимов в Oracle требует привилегий (полномочий) CREATE SYNONYM и CREATE PUBLIC SYNONYM, изначально отсутствующих у пользователя SCOTT. В жизни, чтобы обеспечить пользователя синонимами, не обязательно выдавать ему эти привилегии. Создать пользователю Oracle синоним способен, например, администратор.

Использование синонимов для таблиц — это частный случай. В общем синонимы можно создавать и для некоторых прочих хранимых в БД объектов, например, для процедур или функций.

4.12. Справочная информация о таблицах и прочих объектах в БД

Место хранения справочной информации об объектах, составляющих БД («хранимых в БД»), называется в базах данных *словарем-справочником* (data dictionary). Русский термин-перевод пришел в эту область ИТ из более старой лингвистики, где обозначает «справочную книгу, которая содержит собрание слов (словосочетаний, идиом и т. д.), расположенных по определенному принципу, и дает сведения об их значениях, употреблении, происхождении; информацию о понятиях и предметах, ими обозначаемых и др.». В БД этот термин, не утратив самой общей сути, стал значительно более конкретен. Иногда вместо него пользуются термином *системный каталог*.

В Oracle словарь-справочник реализован набором таблиц в составе самой БД, составляющих «справочную часть» БД. Количество таких таблиц в любой БД Oracle — несколько сотен. Официальный их перечень приведен в документации по Oracle. Вот два примера таблиц словаря-справочника:

- USER_TABLES — перечень всех таблиц схемы пользователя и их одиночных (не множественных) свойств;
- USER_TAB_COLUMNS — перечень столбцов всех таблиц схемы пользователя и одиночных свойств столбцов.

Следующие два запроса к таблицам словаря-справочника предоставляют основные сведения о всех имеющихся в схеме пользователя таблицах и основные сведения о столбцах таблицы PROJ. Три команды COLUMN предназначены для SQL*Plus и задают приемлемый формат выдачи на экран:

```
COLUMN table_name FORMAT A30
```

```
SELECT
    table_name
  , status
FROM
    user_tables
;
```

```
COLUMN column_name FORMAT A30
COLUMN data_type    FORMAT A15
```

```
SELECT
    column_name
  , data_type
  , data_length
  , data_precision
FROM
    user_tab_columns
WHERE
    table_name = 'PROJ'
;
```

Возможности словаря-справочника дополняются способностью Oracle хранить в нем «комментарии», краткие пояснения к сведениям о таблицах и столбцах. Для заведения комментария в Oracle SQL имеется самостоятельная команда COMMENT:

```
COMMENT ON TABLE emp IS 'Общие сведения о сотрудниках';
```

```
COMMENT ON COLUMN emp.comm IS
    'Отсутствие значения есть отсутствие комиссионных'
;
```

```
COMMENT ON COLUMN emp.mgr IS
    'Номер начальника сотрудника берется из столбца EMPNO в EMP'
;
```

Просматривать имеющиеся комментарии можно через таблицы словаря-справочника USER_COL_COMMENTS и USER_TAB_COMMENTS:

```
COLUMN comments FORMAT A30 WORD
```

```
SELECT comments
FROM    user_tab_comments
WHERE   table_name = 'EMP'
;
```

```
SELECT column_name, comments
FROM    user_col_comments
WHERE   table_name = 'EMP'
;
```

Ценность комментирования описаний таблиц и столбцов в справочной части БД, хотя совсем сегодня и не пропала, но уже не столь велика, как в былые времена. Современные инструменты типа SQL Developer Data Modeler позволяют хранить больше пояснительной информации о данных — правда, уже вне словаря-справочника.

5. Общие элементы запросов и предложений DML: выражения

Готовые приложения, как правило, работают с данными уже существующих таблиц. Основная группа предложений SQL, используемых в работающих приложениях, — это SELECT для выборки и операторы DML для изменения данных таблиц. При всем их синтаксическом различии, операторы этой группы роднит использование выражений, составляемых по одним и тем же одинаковым правилам.

В общем случае выражение строится на основе более простых выражений с помощью операторов, функций и прочих конструкций. Началом же для построения любого выражения служат «элементарные», исходные значения, далее не раскрываемые, указанные напрямую тем или иным способом.

5.1. Исходные значения

Исходные значения лежат в основе любого выражения в составе оператора DML или запроса на SQL. Это такие подвыражения, которые при анализе не разложимы далее на другие подвыражения. Исходные значения могут быть сообщены выражению:

- явно,
- через «системные переменные»,
- именем поля строки таблицы.

5.1.1. Явно обозначенные величины («литералы»)

Для явно обозначенных величин (values) в русской литературе часто используется калька с американского английского: «литералы». Оригинальное слово literal представляет собой возникшее со временем в северо-американской литературе жаргонное сокращение от literal value, что дословно означает: напрямую («буквально») указанную в тексте величину (средневековое английское значение слова literal не имеет никакого отношения к компьютерному).

Нелишне помнить, что одни и те же величины часто могут быть обозначены по-разному, например, 1 и +1, и так далее (в известном «треугольнике Фреге» предмет — обозначение — смысл literal value скорее «обозначение»). Для разных видов данных в выражениях предусмотрены разные способы обозначения.

5.1.1.1. Числовые величины

Примеры обозначения целых чисел:

38, +12, -3404

Примеры обозначения «десятичных» чисел (decimal), иначе чисел с возможной дробной частью, записанных в десятичной системе счисления:

342.16, 49, -16, 0.83459

Разделение целой части от дробной осуществляется с помощью десятичной точки или же запятой, в зависимости от установок местности («языковых»). Русский формат записи чисел (в Oracle устанавливается параметром сеанса NLS_NUMERIC_CHARACTERS как ', ') унаследовал исторически французскую традицию употребления в качестве разделителя запятую в отличие от английской точки, унаследованной Северной Америкой как местом разработки СУБД Oracle. Если не обращать внимание на эту мелочь, могут возникать ошибки вывода числовых данных из БД.

В записи рациональных чисел могут присутствовать в служебных целях форматирующие буквы:

49, 18.47, -34e2, 0.16e4, 4e-3

123f (явное указание BINARY_FLOAT),
-123.25d (явное указание BINARY_DOUBLE)

Регистр букв, как обычно, не имеет значения.

Проверить, как Oracle воспринимает указанные в выражениях значения, можно с помощью служебной функции DUMP.

Пример для всех версий:

```
SQL> SELECT 1000, DUMP ( 1000 ) FROM dual;
```

```
      1000 DUMP(1000)
-----
      1000 Тип=2 Len=2: 194,11
```

Пример для версий 10+:

```
SQL> SELECT 123f, DUMP ( 123f ), DUMP ( 123d ) FROM dual;
```

```
      123F DUMP(123F)                                DUMP(123D)
-----
1.23E+002 Тип=100 Len=4: 194,246,0,0 Тип=101 Len=8: 192,94,192,0,0,0,0,0
```

В качестве упражнения предлагается выполнить другие проверки, например, при записи числа как 1000.1 и 1.0e+3.

5.1.1.2. Строки текста

Примеры указания строк:

'Collins',''tis' (кавычки в кавычках), '!?-@ ', ' ', ' ', '1234'

n'Многобайтовая кодировка; по правилам ANSI можно писать и N, и n',
n'Тоже многобайтовая кодировка'

q'[Строка без 'искажений'. Возможно, начиная с версии 10]',
q'|ограничивающий символ может быть практически любой|'

u'строка в Unicode, начиная с версии 10'

Функция DUMP помогает понять, как воспринимает СУБД по-разному оформленные строки. Последние два предложения SELECT работают, начиная с версии 10:

```
SQL> SELECT 'a''bc', DUMP ( 'a''bc' ) FROM dual;
```

```
'A'' DUMP('A''BC')
-----
a'bc Тип=96 Len=4: 97,39,98,99
```

```
SQL> SELECT 'a''bc', DUMP ( q'wa'bcw' ) FROM dual;
```

```
'A'' DUMP(Q'WA'BCW')
-----
a'bc Тип=96 Len=4: 97,39,98,99
```

```
SQL> SELECT 'a''bc', DUMP ( nq'wa'bcw' ) FROM dual;
```

```
'A'' DUMP(NQ'WA'BCW')
-----
a'bc Тип=96 Len=8: 0,97,0,39,0,98,0,99
```

5.1.1.3. Моменты и интервалы времени

Начиная с версии 9 в Oracle поддерживается система указаний моментов и интервалов времени, принятая для SQL комитетами ANSI/ISO по стандартизации SQL. Используемый в примерах ниже формат указания самого значения жестко регламентирован ANSI/ISO. Это касается и типа DATE, для которого Oracle принимает формулировку из стандарта, но по-своему раскрывает ее содержание.

Для обозначения *моментов* времени используются конструкции DATE, TIME и TIMESTAMP. Примеры:

```
DATE '2003-04-14'
    (14 апреля 2003 00:00:00; имеет тип DATE, а временная компонента обнулена)
TIME '12:30:45'
    (12.30.45.000000000 полудни; тип не играет самостоятельной роли в БД и может использоваться
    только в выражении)
TIMESTAMP '2003-04-14 15:16:17'
    (14 апреля 2003 15:16:17.000000000; этот и следующий пример имеет тип TIMESTAMP ( 9 ) )
TIMESTAMP '2003-04-14 15:16:17.88'
    (14 апреля 2003 15:16:17.880000000)
TIMESTAMP '1997-01-31 09:26:56.66 +02:00'
    (31 января 1997 09:26:56.660000000 во второй временной зоне; имеет тип TIMESTAMP ( 9 ) WITH
    TIME ZONE)
TIMESTAMP '1997-01-31 09:26:56.66 Europe/Moscow'
    (31 января 1997 09:26:56.660000000 во временной зоне г. Москвы; имеет тип TIMESTAMP ( 9 ) WITH
    TIME ZONE)
```

Некоторые обозначения вторичны, так как при разборе запроса подменяются СУБД Oracle на другую, более известную конструкцию. Например, DATE '2003-04-14' (как выше) будет заменено на TO_DATE('2003-04-14 00:00:00','yyyy-mm-dd hh24:mi:ss'), то есть функцию преобразования строки текста в DATE по приведенной маске. Формулировка типа DATE '2003-04-14' взята из SQL ANSI/ISO, но наполнена фирмой Oracle собственным содержанием.

Для обозначения *интервалов* времени используются конструкции INTERVAL, допускающие указание подынтервалов «грубого» и «точного» диапазона интервалов и, в дополнение к синтаксису ANSI/ISO, указание точности. Деление на «грубые» и «точные» диапазоны условно. Примеры формулирования «точных» интервалов (диапазон от дней до долей секунд и подынтервалы):

```
INTERVAL '5 04:03:02.01' DAY TO SECOND
    (5 дней, 4 часа, 3 минуты, 2,01 секунды; имеет тип INTERVAL DAY ( 2 ) TO SECOND ( 6 )),
INTERVAL '04:03' HOUR TO MINUTE
    (0 дней, 4 часа, 3 минуты; этот и следующий пример имеет тип INTERVAL DAY ( 2 ) TO SECOND ( 0
    )),
INTERVAL '03' MINUTE
    (0 дней, 0 часов, 3 минуты)
```

В пояснениях в скобках указана точность представления соответствующей компоненты, которая в буквальном указании момента времени не фигурирует, но может быть приведена в описании столбца таблицы.

Примеры формулирования «грубых» интервалов (диапазон из лет и месяцев и два подынтервала):

```
INTERVAL '04-5' YEAR TO MONTH
    (плюс 4 года и 5 месяцев; этот и два следующих примера имеют тип INTERVAL YEAR ( 2 ) TO
    MONTH),
INTERVAL '-4' YEAR
    (минус 4 года),
INTERVAL '5' MONTH
    (плюс 5 месяцев)
```

Во всех перечисленных случаях жесткость формата для указания значения не догматична предельно: ведущие нули и пробелы между лексемами роли не играют. Например, один из предыдущих интервалов с равным успехом мог бы быть записан следующим образом:

```
INTERVAL ' + 4- 0005 ' YEAR TO MONTH
```

5.1.2. «Системные переменные» и «псевдостолбцы»

Оба названия в заголовке заключены в кавычки в силу своей условности.

«Системные переменные» по сути представляют из себя ряд иногда полезных для употребления системных функций без аргументов. Вот некоторые примеры:

<i>Переменная</i>	<i>Тип</i>	<i>Описание</i>
USER	VARCHAR2 (256)	Имя пользователя, выдавшего предложение SQL
UID	NUMBER	Номер пользователя, выдавшего предложение SQL
SYSDATE	DATE	Текущая дата + время суток с точностью до секунды
SYSTIMESTAMP ^[9-]	TIMESTAMP (9) WITH TIME ZONE ^[9-]	Текущая дата + время суток с точностью до 1/100 секунды
DBTIMEZONE ^[9-] и SESSIONTIMEZONE ^[9-]	VARCHAR2 (6) и VARCHAR2 (75)	Зоны времени, определенные для БД и в рамках конкретного сеанса

^[9-] Начиная с версии 9.

Пример использования в выражении:

```
SELECT object_name, owner FROM all_objects WHERE owner <> USER;
```

SYSTIMESTAMP, все же, *может* иметь аргумент, задающий точность долей секунд. В таких случаях называть ее «системной переменной», видимо, не правильно. Сравните результаты:

```
SELECT SYSTIMESTAMP FROM dual;  
SELECT SYSTIMESTAMP ( 2 ) FROM dual;  
SELECT SYSTIMESTAMP ( 0 ) FROM dual;
```

Фигурирующие в документации по Oracle «псевдостолбцы» подобны «системным переменным», но в отличие от них способны давать в запросах на разных строках разные значения, которые вычисляются по мере выполнения определенных фаз обработки запроса и доступны для использования на последующих фазах обработки, образуя как бы дополнительный «столбец». Примеры:

<i>Псевдостолбец</i>	<i>Тип</i>	<i>Описание</i>
ROWNUM	NUMBER	Последовательный номер строки в результате SELECT
LEVEL	NUMBER	Номер уровня выдаваемой строки в предложении SELECT с использованием CONNECT BY
CONNECT_BY_ISCYCLE ^[10-]	NUMBER	В предложении SELECT с использованием CONNECT BY: 1, если потомок узла является одновременно его предком, иначе 0
CONNECT_BY_ISLEAF ^[10-]	NUMBER	В предложении SELECT с использованием CONNECT BY: 1, если узел не имеет потомков
ROWID	VARCHAR2 (256)	Физический адрес строки или хранимого объекта

XMLDATA ^{[9.2-)}	CLOB	Текст документа объекта типа XMLTYPE
OBJECT_ID ^{[10-)}	RAW (16)	Идентификатор объекта в таблице фактических или же виртуальных объектов (в объектном представлении)
OBJECT_VALUE ^{[10-)}	тип объекта	Системное имя для столбца в таблице фактических или же виртуальных объектов (в том числе типа XMLTYPE)
ORA_ROWSCN ^{[10-)}	NUMBER	Порядковый номер изменения в БД (SCN), соответствующий строке таблицы или же блоку данных с этой строкой
COLUMN_VALUE ^{[10.2-)}	тип объекта	Тип элемента результата функций TABLE и XMLTABLE
VERSIONS_STARTSCN ^{[10-)} VERSIONS_STARTTIME ^{[10-)} VERSIONS_ENDSCN ^{[10-)} VERSIONS_ENDTIME ^{[10-)} VERSIONS_XID ^{[10-)} VERSIONS_OPERATION ^{[10-)}		Используются в «быстрых» запросах к прошлым данным (flashback queries)

^{[9.2-)} Начиная с версии 9.2.

^{[10-)} Начиная с версии 10.

^{[10.2-)} Начиная с версии 10.2.

Пример использования в выражениях:

```
SELECT object_name, ROWNUM FROM all_objects WHERE ROWNUM <= 15;
```

Системные переменные (но не «псевдостолбцы») допустимы не только в операторах DML, но также в выражениях в конструкции DEFAULT в описании столбца. Пример тому уже приводился. Еще один источник для величин в конструкции DEFAULT — это так называемый «контекст сеанса», автоматически заводимый СУБД для каждого сеанса клиентской программы. Ниже показано, как с помощью системной функции SYS_CONTEXT в качестве умолчательного значения для поля ENAME строки, добавляемой в таблицу EMP, будет подставляться имя пользователя ОС, под которым ведется работа с Oracle:

```
ALTER TABLE emp MODIFY ( ename
  DEFAULT SUBSTR ( SYS_CONTEXT ( 'USERENV', 'OS_USER' ), 1, 10 )
);
```

Обратите внимание, что максимальная длина имени сотрудника 10 фигурирует в описании EMP дважды: в определении типа столбца и в выражении для DEFAULT. Это благоприятствует будущим недоразумениям, но иного решения SQL не дает.

5.1.3. Величины, взятые из полей строк таблицы

Исходные значения в выражении можно также брать из приведенных в выражении полей записей (строк) таблиц, на которые ссылается предложение SQL. В предыдущем запросе OBJECT_NAME — тривиальное (неразложимое) выражение, составленное из обращения к значению в поле с этим названием в очередной строке таблицы ALL_OBJECTS.

5.2. Составные выражения

Составные выражения построены из более простых выражений путем применения к ним «конструкторов»; в житейском восприятии они, в отличие от исходных значений, собственно и образуют выражения. Они имеют тип, соответствующий типу величины-результата вычисления, то есть «типизированы типом результата».

Ниже приводятся разные варианты конструкторов, позволяющих строить из более простых выражений более сложные:

- операции над числами, строками, моментами и интервалами времени;
- функции;
- операторы CASE;
- скалярные подзапросы.

5.2.1. Арифметические операции и числовые выражения

Числовые выражения можно получить из других числовых выражений с помощью арифметических операции *, /, +, -, и, возможно скобок.

Предположим, что поле COMM в очередной строке имеет значение 25. Тогда справедливы следующие примеры:

<i>Числовое выражение</i>	<i>Значение</i>
6 + 4 * 25	106
0.6e1 + 4 * COMM	106.e0
(50 / 10) * 5	25
1 + '25'	26 (неявное преобразование типа)
123d + 123	246 в формате BINARY_DOUBLE
123d + 123f	246 в формате BINARY_DOUBLE
6 + 4 * COMM	106
(6 + 4) * 25	250
NULL * 30	NULL
1f - BINARY_FLOAT_INFINITY	BINARY_FLOAT_INFINITY
1d + BINARY_DOUBLE_NAN	BINARY_DOUBLE_NAN

Числовая арифметика в SQL несколько отличается от школьной. Вот некоторые особенности.

- Если значение элемента числового выражения отсутствует (то есть помечено как NULL, в обозначениях SQL), значение выражения тоже отсутствует (NULL). Это не всегда привычно для интерпретации «неизвестно какое значение» (которую продвигает SQL). Например:

```
SELECT 1 / 0 FROM dual;
-- Ошибка !
```

```
SELECT NULL / 0 FROM dual;
-- ОК
```

- Тип данных результата приводится к наиболее точному из употребленных в выражении.
- Типы элементов-участников числового выражения должны быть числовыми. Если же встречается строка текста (указанная явно или вычисленная подвыражением), она автоматически приводится к числовому значению, то есть происходит неявное преобразование типа. Фактическое преобразование осуществляется функциями TO_NUMBER, TO_BINARY_FLOAT или TO_BINARY_DOUBLE (в зависимости от контекста), например:

```
SELECT '1' / 2 FROM dual;
```

-- будет обработано как:

```
SELECT TO_NUMBER ( '1' ) / 2 FROM dual;
```

Соответственно, невозможность такого преобразования в конкретных случаях (строка текста несводима к числу) определяется правилами работы этих функций.

Многие специалисты полагают неправильным использование неявного преобразования типов, считая необходимым все преобразования выписывать явно, хотя бы и в ущерб краткости записи. Явное указание преобразования повышает качество кода и снижает риск возникновения ошибок. Кроме этого, в жизни немаловажно, что неявное преобразование не улучшает, а чаще всего снижает эффективность выполнения запроса. Однако, даже если бы у разработчиков Oracle и возникло желание отменить неявное преобразование типов, сделать это уже невозможно, не нарушив обратную совместимость кода. Есть и другое препятствие: функции в Oracle не поддерживают всего допустимого в БД разнообразия типов. Например, отсутствуют целые функции, так что автоматическое преобразование типов становится неизбежным.

5.2.2. Простые выражения над строками текста

Простейшие выражения над строками текста (алфавитно-цифровые, или строковые выражения) можно получить из непосредственно указанных значений и единственной текстовой операцией || («склейки»).

Предположим, что поле ENAME (типа VARCHAR2 (10)) в очередной строке имеет значение 'SMITH'. Тогда справедливы следующие примеры:

<i>Строковое выражение</i>	<i>Значение</i>
'Работник'	Работник (типа CHAR (8))
ENAME	SMITH (типа VARCHAR2 (10))
USER	SCOTT (VARCHAR2 (30))
'зиг' 'заг'	зигзаг
'абв' n'абв' 'абв'	абвабвабв (в многобайтовой кодировке)
'Работник' ENAME	Работник SMITH (типа VARCHAR2 (18))
1845 ' г.'	1845 г.

Как и в случае с числовыми выражениями, здесь происходит неявное преобразования типов: когда в выражении встречается число (возможно, полученное вычислением *числового* подвыражения), оно автоматически переводится в строку текста. Фактическое преобразование осуществляется функцией TO_CHAR. Такое преобразование не может окончиться ошибкой, но не всегда выглядит совсем уж очевидно:

```
SELECT '->' || 001.00 || '<-' FROM dual;
```

-- будет обработано как:

```
SELECT '->' || TO_CHAR ( 001.00 ) || '<-' FROM dual;
```

Возникающие тонкости неявного приведения к строке (в вышеприведенном примере ведущие и незначащие нули в записи числа) следует уточнять по описанию функции TO_CHAR.

5.2.3. Операции над типами «момент» и «интервал времени»

Два вида временных типов — для моментов и для интервалов — имеют свою «временную арифметику», основанную на сложении и вычитании и позволяющую строить простые выражения:

<i>Выражение для времени</i>	<i>Значение</i>
DATE '1990-08-28' + 3	31 августа 1990 года, 00:00:00
3 + DATE '1990-08-28'	31 августа 1990 года, 00:00:00
DATE '1988-12-04' - 5	29 ноября 1988 года, 00:00:00
SYSDATE + 1 / 24	[сейчас] ^(*) плюс час
SYSTIMESTAMP ⁽⁹⁻⁾ - INTERVAL '3' HOUR ⁽⁹⁻⁾	[сейчас] ^(*) минус три часа (типа TIMESTAMP(9) WITH TIME ZONE)
DATE '2005-1-1' - SYSDATE	число нецелых суток до/после Нового 2005 года (типа NUMBER)

TIMESTAMP '2005-1-1 0:0:0' ⁽⁹⁻⁾ - SYSTIMESTAMP ⁽⁹⁻⁾	время до/после Нового 2005 года (типа INTERVAL DAY(9) TO SECOND(9))
SYSDATE - INTERVAL '3' HOUR ⁽⁹⁻⁾	[сейчас] ^(*) минус три часа (типа DATE, т. е. с неявным преобразованием типа)
SYSTIMESTAMP ⁽⁹⁻⁾ - 3 / 24	[сейчас] ^(*) минус три часа (типа DATE, т. е. с неявным преобразованием типа)
DATE '2005-1-1' - SYSTIMESTAMP ⁽⁹⁻⁾	время до/после Нового 2005 года (типа INTERVAL DAY(9) TO SECOND(9) , т. е. с неявным преобразованием типа)

(*) Время компьютера с СУБД.

⁽⁹⁻⁾ Начиная с версии 9.

Примеры употребления:

```
SELECT projno FROM proj WHERE bdate > SYSDATE + 1;
```

```
SELECT * FROM emp
WHERE hiredate >
      ( SELECT hiredate FROM emp WHERE ename = 'JONES' )
      + INTERVAL '2' MONTH
;
```

Упражнение. Проверить значения следующих выражений:

- DATE '2009-01-28' + INTERVAL '1' MONTH
- DATE '2009-01-29' + INTERVAL '1' MONTH
- DATE '2008-01-29' + INTERVAL '1' MONTH
- DATE '2009-01-30' + INTERVAL '1' MONTH

Формат выдачи момента времени можно устанавливать для БД, СУБД и отдельного сеанса. Например, применительно к типу DATE:

```
SQL> SELECT value FROM nls_session_parameters
2> WHERE parameter = 'NLS_DATE_FORMAT';
```

VALUE

```
-----
DD-MON-RR
```

```
SQL> SELECT SYSDATE FROM dual;
```

SYSDATE

```
-----
14-SEP-09
```

```
SQL> ALTER SESSION SET NLS_DATE_FORMAT = 'YYYY-MM-DD HH24:MI:SS';
```

Session altered.

```
SQL> SELECT SYSDATE FROM dual;
```

SYSDATE

```
-----
2009-09-14 00:36:11
```

```
SQL> ALTER SESSION SET NLS_DATE_FORMAT = 'Day HH:MI:SS am';
```

Session altered.

```
SQL> SELECT SYSDATE FROM dual;
```

```
SYSDATE
-----
Monday    12:36:11 am
```

Непосредственно в выражениях формат указывается маской в функциях TO_DATE, TO_TIMESTAMP, TO_CHAR и подобных; обширный перечень способов указать маску приводится в документации по Oracle.

5.2.4. Функции

Средством построения выражений в SQL могут служить функции. В Oracle функции могут быть разных категорий: скалярными, обобщающими (агрегирующими), аналитическими, табличными, общесистемными (встроенными) или написанными пользователями. Далее рассматриваются общесистемные функции, доступные всем пользователям СУБД, разных видов. Полный официальный их перечень из более 200 наименований приведен в документации по Oracle. Часть из них введена в Oracle вслед за стандартом SQL (правда, не всегда с пунктуальным соблюдением правил оформления), а часть — нет.

Самую большую категорию из них составляют скалярные функции, то есть такие, которые принимают скалярные входные значения и вычисляют скалярный ответ. (Скалярность подразумевается здесь в исконном смысле, как признак одиночности величины, в противовес вектору величин. Однако с появлением в Oracle объектных возможностей скалярная величина не обязана быть атомарной, и если эта величина — объект, то она будет иметь известную СУБД структуру).

Ниже приводятся примеры некоторых характерных групп функций.

5.2.4.1. Функции для строк текста

Используются для работы со строками типов VARCHAR2 и CHAR. Примеры функций:

- LENGTH — вычисление длины строки;
- LOWER, UPPER — понижение и повышение регистра букв;
- INITCAP — повышение регистра первых букв в словах и понижение остальных;
- RTRIM, LTRIM — убиение одинаковых символов в конце либо в начале (по умолчанию пробелов);
- RPAD, LPAD — дополнение строки текста одинаковыми символами справа либо слева (по умолчанию пробелами);
- TRANSLATE — подстановка символов в строке;
- INSTR, SUBSTR — поиск места указанной подстроки и поиск подстроки по месту.

Примеры действия функций на строки:

```
SQL> SELECT ename, LOWER ( ename ), INITCAP ( ename ) FROM emp;
```

ENAME	LOWER (ENAM	INITCAP (EN
-----	-----	-----
SMITH	smith	Smith
ALLEN	allen	Allen
...		

```
SQL> SELECT LPAD ( ename, 7, '*' ), RTRIM ( ename, 'ITH' ) FROM emp;
```

LPAD (EN	RTRIM (ENAM
-----	-----
**SMITH SM	
**ALLEN ALLEN	
***WARD WARD	
...	

```
SQL> SELECT TRANSLATE ( 'Объём', 'ё0', 'eO' ) FROM dual;
```

```
TRANS
-----
Объём
```

Большую часть все эти функции в Oracle определены в стандарте SQL, а значит, будут доступны и будут исполняться тем же порядком в СУБД ряда прочих типов.

5.2.4.2. Функции преобразования типов данных

В соответствии со стандартом SQL-92 в Oracle есть общая функция преобразования типов CAST. Примеры:

```
SELECT CAST ( '0123' AS NUMBER ( 5 ) ) FROM dual;
SELECT CAST ( SYSDATE AS VARCHAR2 ( 20 ) ) FROM dual;
```

Она способна выполнять большинство преобразований, имеющих смысл. Однако в Oracle эта функция используется нечасто (за исключением преобразования типов коллекций, объяснение которых смотри ниже) ввиду наличия собственных, более развитых её замен:

```
TO_CHAR
TO_CLOB
TO_NUMBER
TO_BINARY_FLOAT/DOUBLE
TO_DATE
TO_TIMESTAMP
TO_YMINTERVAL, TO_DSINTERVAL
NUMTOYMINTERVAL, NUMTODSINTERVAL
других.
```

Большинство из них имеет имена, начинающиеся с 'TO_', однако Oracle не пунктуальна в соблюдении этого неформального правила.

Обычно эти функции допускают уточнение способа преобразования, с помощью маски или географических установок местности. Примеры употребления:

```
SELECT TO_TIMESTAMP ( '10-APR-56' ) FROM dual;
```

```
SELECT
  TO_TIMESTAMP (
    '10-Апрель-56'
  , 'DD-MONTH-RR'
  , 'NLS_DATE_LANGUAGE=RUSSIAN'
  )
FROM dual;
```

```
SELECT TO_CHAR ( SYSDATE, 'Day HH24:MI:SS' ) FROM dual;
```

Использование маски позволяет в частности поставить под контроль выдачу номера недели. В США, где разрабатывается СУБД Oracle, неделя начинается с воскресения, а недели отсчитываются с первого дня года. По правилам ISO это не так. Правильно выбранная маска способна заставить СУБД выдать желаемое. Вот пояснительная пара запросов со сравнительной выдачей:

```
COLUMN "Неделя в США"   FORMAT A13
COLUMN "Неделя по ISO"  FORMAT A13
COLUMN "Название дня"   FORMAT A13

SELECT
  TO_CHAR ( DATE '2010-1-1', 'ww' )  "Неделя в США"
```

```
, TO_CHAR ( DATE '2010-1-1', 'iw' ) "Неделя по ISO"
, TO_CHAR ( DATE '2010-1-1', 'day' ) "Название дня"
FROM dual
;
```

```
SELECT
  TO_CHAR ( DATE '2010-1-4', 'ww' ) "Неделя в США"
, TO_CHAR ( DATE '2010-1-4', 'iw' ) "Неделя по ISO"
, TO_CHAR ( DATE '2010-1-5', 'day' ) "Название дня"
FROM dual
;
```

Использование других масок преобразования, приведенных в документации по Oracle, помогает справиться с другими неоднозначностями, возникающими при работе со временем.

5.2.4.3. Функции для работы со временем

Позволяют пополнить «временную арифметику» необходимыми или практичными операциями. Примеры функций:

```
ADD_MONTHS
LAST_DAY
MONTHS_BETWEEN
NEXT_DAY
ROUND
TRUNC
EXTRACT[9-]
```

^[9-] Начиная с версии 9.

Допустим, что в поле BDATE типа DATE текущей строки находится значение «5 сентября 1999 г., 13 часов 30 минут 05 секунд». Справедливы следующие оценки выражений:

Выражение	Значение
EXTRACT (DAY FROM SYSTIMESTAMP)	[сегодняшний день] ^(*) (типа NUMBER)
EXTRACT (HOUR FROM SYSTIMESTAMP)	[время суток] ^(*) (типа NUMBER)
EXTRACT (MINUTE FROM INTERVAL '04:03' HOUR TO MINUTE)	и так далее 3 «минуты» (типа NUMBER)
TRUNC (BDATE)	5 сентября 1999 года, 00:00:00
TRUNC (BDATE, 'year')	1 января 1999 года, 00:00:00
TRUNC (BDATE, 'dd')	5 сентября 1999 года, 00:00:00
TRUNC (BDATE, 'day')	30 августа 1999 года, 00:00:00
TRUNC (BDATE, 'hh')	5 сентября 1999 года, 13:00:00
ADD_MONTHS (BDATE, 2)	5 ноября 1999 года, 13:30:05
(ADD_MONTHS (TRUNC (SYSDATE, 'year'), 12) - 1) - TRUNC (SYSDATE)	число суток до ближайшего Нового года (типа NUMBER)

^(*) Время компьютера с СУБД.

Можно заметить, что Oracle SQL еще менее последователен в своем синтаксисе, нежели стандарт SQL. Сравните указание компоненты момента времени в виде строки текста ('year' в функции TRUNC, а также ROUND) и с помощью ключевого слова (DAY, HOUR в функции EXTRACT).

Отход от традиционных обозначений функции как $f(x, \dots)$ и включение в обращение к функции разного рода ключевых слов характерны для Oracle (как и для стандартного SQL), и активно укрепляются от версии к версии последнее время. Другие примеры встретятся далее. Смысл такого оформления, очевидно, в стремлении улучшить читаемость текстов запросов и использовать синтаксический анализатор для проверки правильности употребления типов данных. Насколько это оправдывает себя, программист волен судить самостоятельно.

Примеры использования:

```
SELECT EXTRACT ( MONTH FROM INTERVAL '123-2' YEAR ( 4 ) TO MONTH )
FROM dual
;

SELECT EXTRACT ( MONTH FROM INTERVAL '-123-2' YEAR ( 4 ) TO MONTH )
FROM dual
;

SELECT EXTRACT ( MONTH FROM SYSTIMESTAMP ) FROM dual;

SELECT EXTRACT ( YEAR FROM SYSTIMESTAMP ) FROM dual;

SELECT MONTHS_BETWEEN ( DATE '2009-03-01', DATE '2009-02-28' )
FROM dual
;
```

Заметьте, что функции «месячной арифметики» ADD_MONTHS и MONTHS_BETWEEN вовсе не так очевидны, как могли бы показаться.

Упражнение. Проверить значения следующих выражений:

- ADD_MONTHS (DATE '2009-01-28', 1)
- ADD_MONTHS (DATE '2009-01-29', 1)
- ADD_MONTHS (DATE '2008-01-29', 1)
- ADD_MONTHS (DATE '2009-01-30', 1)

Упражнение. Проверить значения следующих выражений:

- ADD_MONTHS (DATE '2008-02-28', 3)
- ADD_MONTHS (DATE '2008-02-29', 3)

5.2.4.4. Функции условной подстановки значений

Дают возможность выполнить «преобразование» аргументов, а по сути — условную замену конкретных величин. Часть таких функций связана с желательной для программиста переработкой отсутствующих значений (в отдельном случае Not a Number), а функция DECODE — нет:

Функция	Логический эквивалент
NVL (E ₁ , E ₂)	IF E ₁ IS NULL THEN E ₂ ELSE E ₁
NVL2 (E ₁ , E ₂ , E ₃)	IF E ₁ IS NULL THEN E ₃ ELSE E ₂
NANVL (E ₁ , E ₂) ^(IEEE 754)	IF E ₁ IS NAN THEN E ₂ ELSE E ₁
COALESCE (E ₁ , E ₂ , E ₃ , ...) ⁽⁹⁻⁾	первое по списку E _i со значением, не NULL
DECODE (E ₁ , E ₂ , E ₃ , ...[, E _N])	IF E ₁ = E ₂ THEN E ₃ [ELSE IF E ₁ = E ₄ THEN E ₅ [...]] [ELSE E _N]

^(IEEE 754) Для типов BINARY_FLOAT/BINARY_DOUBLE.

⁽⁹⁻⁾ Начиная с версии 9.

Примеры:

```
SELECT ename, comm, NVL ( comm, 0 ) FROM emp;

SELECT comm, sal, COALESCE ( comm, sal ) FROM emp;
```

В отличие от NVL и NVL2, COALESCE не вычисляет выражения-аргументы без надобности:

```
SELECT NVL ( 123, 1 / 0 ) FROM dual;
```

-- Ошибка !

```
SELECT COALESCE ( 123, 1 / 0 ) FROM dual;
```

-- OK

Пример DECODE:

```
SELECT
  deptno
, loc
, DECODE ( loc, 'NEW YORK', 'NEW YORK CITY', 'BOSTON', 'BOSTON AREA' )
FROM dept
;
```

Фактически DECODE позволяет сформулировать в тексте запроса таблицу подстановки значений. В нашем случае, если потребуется выдать исходное значение LOC, когда там не значения 'NEW YORK' и 'BOSTON', нужно будет добавить замыкающий четный аргумент:

```
DECODE
( loc, 'NEW YORK', 'NEW YORK CITY', 'BOSTON', 'BOSTON AREA', loc )
```

Это не самое хорошее решение, так как иногда приходится повторять сложное выражение вторично, что чревато ошибками и лишними вычислениями. Кроме того, методически оправдано держать правила преобразования в БД, а не в тексте запроса (если только эти правила имеют прикладное значение). На практике же нахождение таблицы преобразования в БД резко замедлит вычисление.

5.2.4.5. Нескалярные функции для анализа данных

Таковых имеется две категории: «агрегатные», то есть обобщающие, и «аналитические».

Агрегатные функции иначе называют «стандартными агрегатными» функциями и «агрегирующими» функциями. До версии 8.1.6 они (сокращенным количеством) назывались «статистическими». Они дают скалярный результат, но аргументом им служит столбец значений, чем конструктивно отличаются от большинства других встроенных функций. Такое устройство сообщает им обобщающий характер.

Некоторые из них:

Функция	Описание
COUNT	Число строк со значениями в столбце или строк в таблице
MIN	Наименьшее значение в столбце
MAX	Наибольшее значение в столбце
SUM	Сумма значений в столбце
AVG	Среднее арифметическое значений в столбце
VARIANCE	Дисперсия (мера отклонения от математического ожидания)
STDDEV, STDDEV_POP, STDDEV_SAMP	Стандартное отклонение (квадратный корень от дисперсии) в разных вариациях
MEDIAN ^[10-]	Медиана значений в столбце
LISTAGG ^[11.2-]	Склейка значений в группе посредством указанного разделителя

^[10-] С версии 10.

^[11.2-] С версии 11.2.

Некоторые другие примеры: CORR, COVAR_POP, COVAR_SAMP, CUME_DIST, PERCENTILE_COUNT и так далее.

Начиная с версии 10 Oracle позволяет производить более «серьезные» статистические обобщения данных столбца таблицы, но уже не средствами SQL, а программно, с помощью процедур из встроенного пакета DBMS_STAT_FUNCS. Они позволяют определить соответствие указанного значения тому или иному виду

статистического распределения, а также обобщать данные столбца всеми способами стандартных агрегатных функций, но вдобавок со значительным количеством дополнительной обобщающей информации.

Аналитические функции идут дальше агрегатных, не только имея столбцовые аргументы, но и возвращая в виде столбца результат. Они не только позволяют обобщить данные, как агрегатные, но способны делать это без потери детализации.

Агрегатные и аналитические функции отличаются от скалярных по формальному употреблению в тексте запроса и требуют в силу этого отдельного рассмотрения, которое последует в соответствующих разделах описания предложения SELECT ниже.

5.2.5. Конструкции (операторы) CASE для построения выражений

В качестве альтернативы функции DECODE (отсутствующее в стандарте решение Oracle, оформленное в виде функции), и других функций условной подстановки значений NVL, NVL2, NANVL и COALESCE, начиная с версии 8.1.6 можно пользоваться «поисковым» CASE-выражением, а с версии 9 — «простым» CASE-выражением (оба входят в стандарт SQL-92). Формально конструкцию CASE можно считать оператором (с более сложной синтаксической структурой, нежели в случае, положим, арифметических операторов), предназначенным для построения выражений из более простых. Для употребления существенно, что результат CASE не «окончателен»; он представляет из себя выражение, которое не возбраняется использовать для построения очередного более сложного. В этом конструкция CASE не отличается от прочих операторов.

Синтаксис «поискового» оператора CASE:

```
CASE
  WHEN условное-выражение1 THEN выражение-результат1
  WHEN условное-выражение2 THEN выражение-результат2
  ...
  WHEN условное-выражениеN THEN выражение-результатN
  [ ELSE выражение-результат ]
END
```

Проверки происходят сверху вниз, пока первое по порядку *условное-выражение*_i не станет TRUE. Тогда проверки прекратятся, и результатом CASE будет значение *выражения-результата*_i.

Синтаксис «простого» оператора CASE:

```
CASE выражение0
  WHEN выражение1 THEN выражение-результат1
  WHEN выражение2 THEN выражение-результат2
  ...
  WHEN выражениеN THEN выражение-результатN
  [ ELSE выражение-результат ]
END
```

Проверки происходят сверху вниз, пока значение первого по порядку *выражения*_i не станет равным значению *выражения*₀. Тогда проверки прекратятся, и результатом CASE будет значение *выражения-результата*_i.

Все выражения допускается заключать в скобки, играющие чисто оформительскую роль. Следующие два выражения равносильны:

```
CASE
WHEN EXTRACT ( MONTH FROM SYSDATE ) BETWEEN 3 AND 5 THEN 'Весна'
ELSE 'Непонятно что'
END
```



```

CASE
WHEN ( EXTRACT ( MONTH FROM SYSDATE ) BETWEEN 3 AND 5 ) THEN ( 'Весна' )
ELSE ( 'Непонятно что' )
END

```

Синтаксис *условного-выражения* в CASE соответствует синтаксису подобного в части WHERE предложений SELECT, UPDATE и DELETE, описываемых далее, и допускает достаточно сложные конструкции, как показывает пример ниже:

```

SELECT
    ename
,   sal
,   deptno
,   CASE
    WHEN sal > 4000
        THEN 'Highly paid'
    WHEN deptno IN ( SELECT deptno FROM dept WHERE loc = 'NEW YORK' )
        THEN 'Works in New York'
    ELSE 'Nothing interesting'
    END || ' !'
    attention
FROM emp
;

```

Заметьте, что по нашим данным в результате служащий KING будет помечен как «высокооплачиваемый». Если в операторе CASE проверку зарплаты и местонахождения отдела поменять местами, KING окажется помечен как «работающий в Нью-Йорке».

Упражнение. Проверьте последнее утверждение.

Тем самым, конструкция CASE вносит элемент процедурности в описательное в целом построение запроса, принятое в SQL.

Отсутствие конструкции ELSE может приводить к отсутствию значения в результате (к NULL), однако же не к ошибке:

```

SQL> SELECT NVL ( CASE 1 WHEN 2 THEN 3 END, -1 ) FROM dual;

NVL(CASE1WHEN2THEN2END,-1)
-----
-1

```

Существует мнение, что *обязательное* указание ELSE улучшает понимание текста программистом (а значит снижает риск человеческих ошибок). Согласно этой точке зрения следующее выражение *не* является построенным удачно:

```

CASE loc
WHEN 'NEW YORK' THEN 'NEW YORK CITY'
WHEN 'BOSTON'   THEN 'BOSTON AREA'
END

```

Вместо этого лучше написать:

```

CASE loc
WHEN 'NEW YORK' THEN 'NEW YORK CITY'
WHEN 'BOSTON'   THEN 'BOSTON AREA'
ELSE NULL
END

```

«Поисковая» разновидность CASE носит более общий характер, нежели «простая», так как допускает условные выражения, которые получены операторами сравнения, отличными от = (равенства).

Из-за того, что конструкция CASE оформлена в виде оператора языка, а не функции, как DECODE, NVL, NVL2, NANVL и COALESCE, она становится не только их более общим заменителем, но к тому же и быстрее их вычислимой, хотя бы и ненамного в каждом отдельном случае. Это создает стимул к применению в программировании именно ее, а не перечисленных функций условной подстановки значений. В то же время тексте запроса она обычно занимает больше места.

5.2.6. Скалярный запрос

Еще одна конструкция для формирования выражений существует с версии Oracle 9. Если запрос одностолбцовый и возвращает не более одной строки, его можно указать в круглых скобках в составе выражения на правах значения. Пример:

```
SELECT
  ename
, '-> ' || ( SELECT dname FROM dept WHERE dept.deptno = emp.deptno )
FROM emp
WHERE
  TRUNC ( hiredate, 'day' ) >=
  TRUNC
    ( ( SELECT hiredate FROM emp WHERE job = 'PRESIDENT' ), 'day' )
;
```

(Сотрудники, поступившие на работу на одной неделе с президентом, или позже.)

При этом множественный результат воспринимается как ошибка, а пустой результат — как отсутствие значения, NULL:

```
SELECT
  ename
, ( SELECT deptno FROM dept WHERE 1 = 2 ) + 0
FROM emp
;
```

Добавление нуля в выражении выше сделано, чтобы убедить читателя в отсутствии значения у приведенного скалярного выражения. Иначе подошло бы использование функции NVL или оператора CASE.

Упражнение. Переписать последний запрос с использованием функции NVL для выяснения реакции СУБД на отсутствия строк в скалярном запросе.

Одностолбцовость скалярного запроса Oracle в состоянии контролировать синтаксически, а вот однострочность — нет. Для повышения надежности текста некоторые предлагают в качестве искусственной меры включать в условное выражение во фразе WHERE запроса дополнительное условие ROWNUM <= 1, например:

```
( SELECT hiredate FROM emp WHERE job = 'PRESIDENT' AND ROWNUM <= 1 )
```

Не исключено, что такая мера более важна как способ привлечения внимания программиста к содержательно правильному построению запроса, и такое дополнительное условие служит своего рода «активным комментарием» к тексту программы. Обратите внимание, что добавление AND ROWNUM <= 1 несколько изменяет смысл запроса.

Скалярный подзапрос скалярен в том же смысле, что и упоминавшиеся скалярные функции, то есть результат его не может быть массивом (например, столбцом значений). В то же время единственное возвращаемое им значение вполне может быть объектом (в смысле объектных возможностей Oracle) и иметь понятную СУБД структуру.

5.2.7. Условные выражения

Условные выражения в Oracle существуют, но в отличие числовых, строковых и временных не могут использоваться для придания значений полям строк таблиц БД, так как в Oracle отсутствует тип BOOLEAN (хотя он есть в стандарте SQL:1999). Не будучи в той же степени равными, они активно используются для проверки условия в операторе CASE (см. выше), а также в части START WITH фразы CONNECT BY и во фразах WHERE и HAVING предложений SELECT, UPDATE, DELETE (см. ниже).

5.3. Отдельные замечания по поводу отсутствия значения в выражениях

Выражение с операндом, значение которого отсутствует (в явной форме указанного как NULL), приведет к отсутствующему же значению (NULL) в случае:

- числовых и временных выражений, построенных арифметическими операциями, и
- сравнения выражений всех видов.

Понять обработку NULL помогает следующее правило: SQL воспринимает NULL в нетекстовых выражениях как *неизвестное* значение.

Это правило не соблюдается последовательно, и имеет два исключения. Например, числовые выражения $NULL / 0$ и $NULL * 0$ оцениваются как NULL, то есть, «неизвестно что» (в понимании SQL) не есть «все, что угодно». По логике «любое число» («все, что угодно») надо бы в первом случае возвращать ошибку, а во втором — 0; но разработчики стандарта SQL этого делать не стали.

В то же время, если исходить из формальной точки зрения SQL — дополнительное значение для всех типов — или Oracle SQL — дополнительное значение для каждого типа по отдельности, — то в подобной оценке $NULL / 0$ и $NULL * 0$ нет исключений, а есть обычное формальное соглашение.

При работе с отсутствующими значениями в БД часто используют функцию NVL. Сравните ответы:

(1) SELECT ename, sal, comm, sal + comm FROM emp;

и:

(2) SELECT ename, sal, comm, sal + NVL (comm, 0) FROM emp;

В случае (1) получим:

ENAME	SAL	COMM	SAL+COMM
SMITH	800		
ALLEN	1600	300	1900
WARD	1250	500	1750
JONES	2975		
MARTIN	1250	1400	2650
BLAKE	2850		
CLARK	2450		
SCOTT	3000		
KING	5000		
TURNER	1500	0	1500
ADAMS	1100		
JAMES	950		
FORD	3000		
MILLER	1300		

В случае (2) получим:

ENAME	SAL	COMM	SAL+NVL (COMM, 0)
SMITH	800		800

ALLEN	1600	300	1900
WARD	1250	500	1750
JONES	2975		2975
MARTIN	1250	1400	2650
BLAKE	2850		2850
CLARK	2450		2450
SCOTT	3000		3000
KING	5000		5000
TURNER	1500	0	1500
ADAMS	1100		1100
JAMES	950		950
FORD	3000		3000
MILLER	1300		1300

К сожалению, формального обоснования применению функции NVL в подобных случаях не существует. Стоит ее употребить или нет, решается смыслом, который проектировщик БД закладывает в допущение пропуска значения в столбце. В нашем случае, если смысл — «комиссионные неизвестны» (unknown, «значение отсутствует, потому что не известно базе данных, не поступило в БД»), то следует применить запрос (1). Если же смысл «комиссионных нет» («сотрудник не получил комиссионных»), то запрос (2). Смысл пропущенного значения в таблице SQL никак не означен в БД; он существует вне БД, однако же должен учитываться в программе, работающей с БД. Это одна из давно известных неприятностей SQL.

Частично решить именно эту проблему можно было бы использованием вместо одного «безликого» признака отсутствия значения NULL хотя бы двух с разным смыслом (предлагалось «неприменимо» — missing but inapplicable — и «неизвестно» — missing but applicable). Однако в этом случае возникли бы другие проблемы, связанные со сложностью употребления четырехзначной логики, и по этой причине в SQL от этого отказались. Разработчики SQL советуют использовать пропущенные значения в столбцах только в смысле unknown = missing but applicable. В Oracle этот совет имеет относительную ценность, так как некоторые запросы (примеры встретятся далее) способны порождать пропущенные значения именно в смысле missing but inapplicable.

Полным же решением мог стать отказ от отсутствующих значений вообще. Поскольку в SQL этого не сделано, некоторые советуют добровольно избегать употребления отсутствующих значений по мере возможности и моделировать отсутствие значений (в силу разных причин) без использования NULL. Обратной стороной такого самоограничения окажется загромождение схемы данных и усложнение запросов к БД.

6. Выборка данных

*Бесқозырқа белая, в полоску воротник..
Пионеры смелые спросили напрямик:
«С қақого, парень, года, с қақого парохода
И на қақих морях ты побывал, моряк?»*

Артековская песня, слова З. Александровой, обработка В. Моделя

6.1. Фразы предложения SELECT

Предложение SELECT в SQL складывается из фрагментов, которые по примеру лингвистики носят название *фраз* (clauses). Иногда их удобно называть более общим словом «конструкции» или же словами «часть предложения».

Допустимые в предложении SQL фразы — это: SELECT, FROM, WHERE, GROUP BY, HAVING, CONNECT BY, ORDER BY, PIVOT/UNPIVOT, FETCH ROWS. Общие правила использования фраз в предложении SELECT следующие.

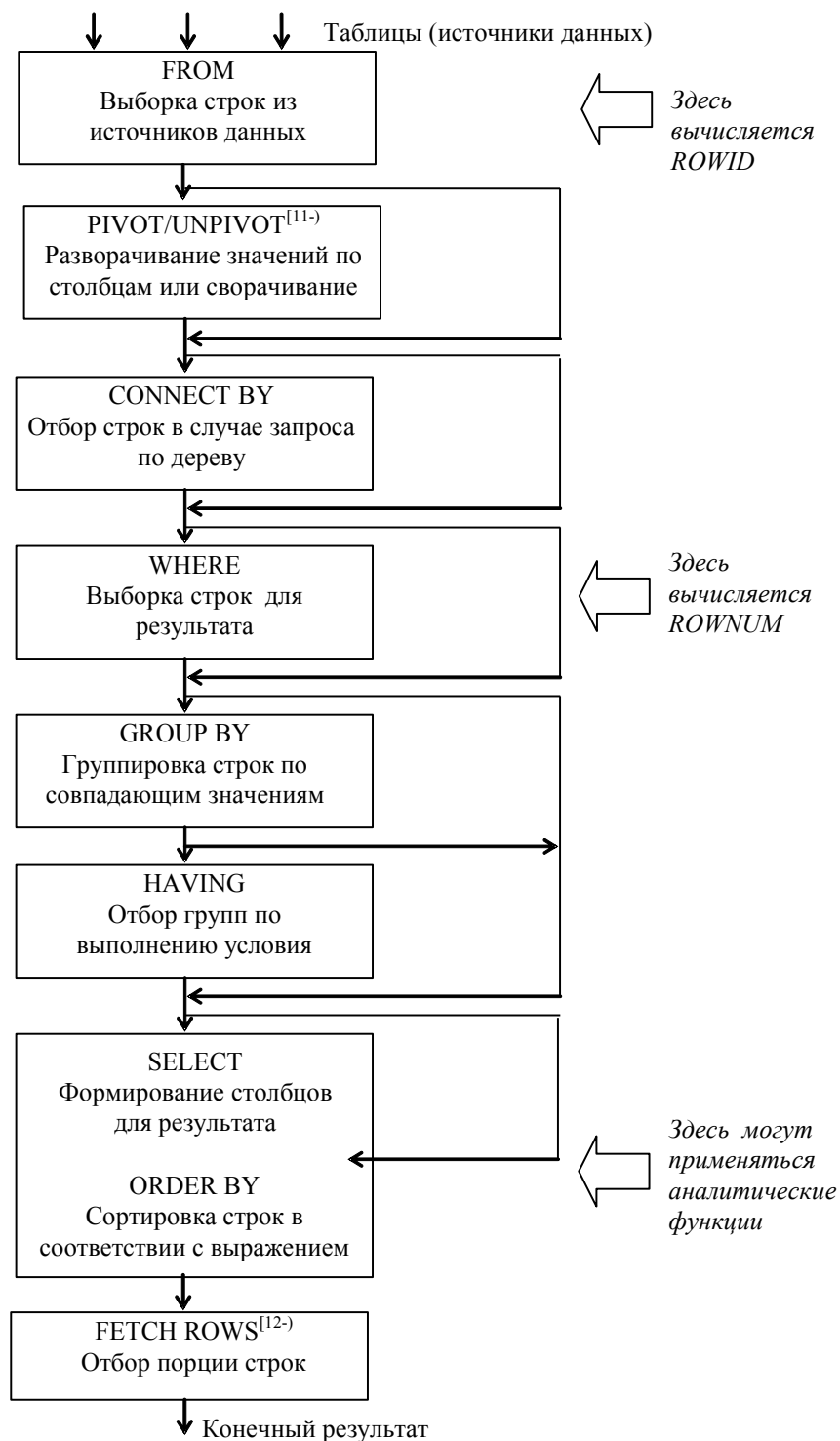
- В каждом предложении обязаны быть фразы SELECT и FROM. Остальные фразы необязательны.
- Порядок следования фраз во всех предложениях фиксирован (например, GROUP BY всегда следует за WHERE и FROM, а ORDER BY всегда стоит в конце).
- Фраза HAVING может употребляться только в в дополнении ко GROUP BY.

Начиная с версии 10 в Oracle SQL возможно еще употребление фразы MODEL, которая, однако, стоит особняком от прочих в силу своей синтаксической инородности и нетрадиционности для SQL. Сфера ее применения также специфична — это базы типа «склады данных», data warehouses. Ниже она не рассматривается.

Кроме этого может иметься фраза WITH, которую можно рассматривать как оформительскую (с версии 9) и процедурную (с версии 11.2) надстройку над «традиционным» предложением SELECT. Она будет рассмотрена ниже в соответствующем разделе.

6.1.1. Логический порядок обработки предложения SELECT

Порядок обработки предложения SELECT также фиксирован и почти совпадает с порядком написания фраз в тексте. Исключения составляет (а) фраза SELECT, которая в отличие от написания обрабатывается в последнюю очередь, и (б) фразы CONNECT BY и WHERE, которые обрабатываются в порядке, обратном написанию:



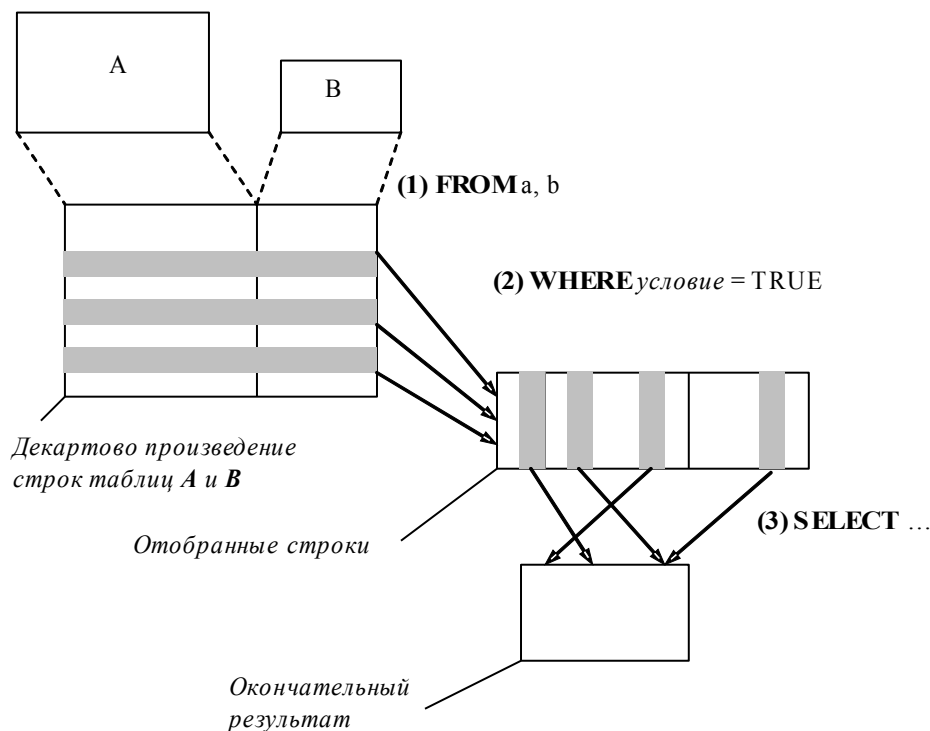
^{[11-)} С версии 11.

^{[12-)} С версии 12.

Указанный порядок — логический. Это значит, что технические планы обработки запросов СУБД может предлагать самые разнообразные, и вовсе не обязательно они будут точно воспроизводить приведенную выше последовательность в каждом отдельном случае. Причина в чудовищной затратности, которой чревато простое следование вышеприведенной схеме в самых рядовых запросах. Однако действие любого плана, фактически предлагаемого Oracle, никогда не будет вступать в противоречие с этой логикой обработки. Иными словами, когда программисту требуется понять, каким может оказаться

результат запроса, он вполне вправе положиться на эту общую логическую последовательность обработки и не вдаваться в подробности фактического плана выполнения.

Она основывается на чередовании этапов вычисления (очередная фраза), так что каждый этап принимает какое-то множество данных на входе и вырабатывает множество данных на выходе. За исключением двух крайних случаев: множества данных на входе фразы FROM и окончательного результата на выходе фразы SELECT, — такие множества можно назвать промежуточными результатами вычислений в предложении SELECT. Вот как выглядит логическая схема обработки предложения SELECT, пожалуй, самого распространенного вида SELECT ... FROM a, b WHERE условие:



В частности, эта схема делает очевидным отсутствие влияния порядка указания источников данных во фразе FROM на результат (в понимании реляционной модели, то есть, если забыть о предполагаемом в SQL порядке столбцов, который здесь может проявить себя в случае SELECT * FROM ...).

Далее приводятся два примера предсказания ответа путем применения техники последовательности промежуточных вычислений.

6.1.2. Пример 1 предложения SELECT

«Выдать в порядке приема на работу всех продавцов с их окладами»:

```
SELECT  ename, sal
FROM    emp
WHERE   job = 'SALESMAN'
ORDER BY hiredate
```

(Здесь предполагается, что данные о *всех* сотрудниках исчерпывается таблицей EMP. В базе, отличной от схемы SCOTT, где не соблюдена ортогонализация данных, это может оказаться и не так).

6.1.2.1. Промежуточный результат после фраз FROM и WHERE

В силу того, что источник данных для запроса в данном случае один, фраза FROM фактически ничего не вычисляет, а просто «берет» данные таблицы EMP из базы. Реальную работу — отсев строк согласно условию — выполняет фраза WHERE:

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	300	30
7521	WARD	SALESMAN	7698	22-FEB-81	1250	500	30
7654	MARTIN	SALESMAN	7698	28-SEP-81	1250	1400	30
7844	TURNER	SALESMAN	7698	08-SEP-81	1500	0	30

6.1.2.2. Промежуточный результат после фразы ORDER BY

На множестве строк, полученных от WHERE, фраза ORDER BY наводит порядок:

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	300	30
7521	WARD	SALESMAN	7698	22-FEB-81	1250	500	30
7844	TURNER	SALESMAN	7698	08-SEP-81	1500	0	30
7654	MARTIN	SALESMAN	7698	28-SEP-81	1250	1400	30

6.1.2.3. Конечный отбор фразой SELECT

Из упорядоченного множества строк, полученных от ORDER BY, фраза SELECT формирует столбцы окончательного ответа:

ENAME	SAL
ALLEN	1600
WARD	1250
TURNER	1500
MARTIN	1250

Заметьте, что в ответ не попали значения столбца HIREDATE, по которым упорядочены строки.

6.1.3. Пример 2 предложения SELECT

«Выдать всех *работающих* сотрудников с названиями их отделов»:

```
SELECT ename, dname
FROM emp, dept
WHERE emp.deptno = dept.deptno
```

(Запрос не выдаст сотрудников, не причисленных ни к одному отделу, однако в наших данных таковых нет).

6.1.3.1. Промежуточный результат после фразы FROM

При количестве источников данных более двух фраза FROM делает реальную работу и в нашем случае даст следующий результат:

EMPNO	ENAME	...	DEPTNO	DEPTNO	DNAME	LOC
7369	SMITH	...	20	10	ACCOUNTING	NEW YORK

7499	ALLEN	...	30	10	ACCOUNTING	NEW YORK
7521	WARD	...	30	10	ACCOUNTING	NEW YORK
7566	JONES	...	20	10	ACCOUNTING	NEW YORK
7654	MARTIN	...	30	10	ACCOUNTING	NEW YORK
7876	ADAMS	...	20	10	ACCOUNTING	NEW YORK
7900	JAMES	...	30	10	ACCOUNTING	NEW YORK
7902	FORD	...	20	10	ACCOUNTING	NEW YORK
7934	MILLER	...	10	10	ACCOUNTING	NEW YORK
7369	SMITH	...	20	20	RESEARCH	DALLAS
7499	ALLEN	...	30	20	RESEARCH	DALLAS
7521	WARD	...	30	20	RESEARCH	DALLAS
... ..						
7876	ADAMS	...	20	40	OPERATIONS	BOSTON
7900	JAMES	...	30	40	OPERATIONS	BOSTON
7902	FORD	...	20	40	OPERATIONS	BOSTON
7934	MILLER	...	10	40	OPERATIONS	BOSTON

Такой результат работы фразы FROM является декартовым произведением строк таблиц EMP и DEPT, то есть множеством всех возможных комбинаций строк этих таблиц друг с другом.

Серым фоном отмечены строки, далее отбираемые из этого промежуточного результата на следующем шаге.

6.1.3.2. Промежуточный результат после фразы WHERE

Из полученного от FROM множества строк фраза WHERE отфильтрует для дальнейшей обработки следующие:

EMPNO	ENAME	...	DEPTNO	DEPTNO	DNAME	LOC

7369	SMITH	...	20	20	RESEARCH	DALLAS
7499	ALLEN	...	30	30	SALES	CHICAGO
7521	WARD	...	30	30	SALES	CHICAGO
7566	JONES	...	20	20	RESEARCH	DALLAS
7654	MARTIN	...	30	30	SALES	CHICAGO
7698	BLAKE	...	30	30	SALES	CHICAGO
7782	CLARK	...	10	10	ACCOUNTING	NEW YORK
7788	SCOTT	...	20	20	RESEARCH	DALLAS
7839	KING	...	10	10	ACCOUNTING	NEW YORK
7844	TURNER	...	30	30	SALES	CHICAGO
7876	ADAMS	...	20	20	RESEARCH	DALLAS
7900	JAMES	...	30	30	SALES	CHICAGO
7902	FORD	...	20	20	RESEARCH	DALLAS
7934	MILLER	...	10	10	ACCOUNTING	NEW YORK

Серым фоном отмечены столбцы, далее отбираемые из этого промежуточного результата на следующем шаге.

6.1.3.3. Конечный отбор фразой SELECT

Из полученного от WHERE множества строк фраза SELECT сформирует столбцы окончательного ответа:

ENAME	DNAME

SMITH	RESEARCH
ALLEN	SALES
WARD	SALES
JONES	RESEARCH
MARTIN	SALES
BLAKE	SALES
CLARK	ACCOUNTING

SCOTT	RESEARCH
KING	ACCOUNTING
TURNER	SALES
ADAMS	RESEARCH
JAMES	SALES
FORD	RESEARCH
MILLER	ACCOUNTING

Завершающие наблюдения:

- строки окончательного ответа не считаются упорядоченными (в отличие от предшествовавшего запроса), так что повторение этого же запроса формально может вернуть их теми же, но в другой последовательности;
- порядок перечисления таблиц во фразе FROM (в нашем случае <EMP, DEPT>, а не <DEPT, EMP>) никак не сказывается на конечном результате.

6.2. Логическая целостность обработки предложением SELECT

В процессе обработки СУБД предложения SELECT источники данных, используемые этим предложением, могут подвергаться изменениям, например, со стороны других работающих программ. Если не принимать специальных мер, такие «посторонние» изменения запрашиваемых таблиц могут непредсказуемо исказить результат или даже приводить к конфликтам вычислений.

СУБД Oracle обеспечивает целостность выполнения каждого предложения SELECT, изолируя его от последствий подобных чужих изменений. Иными словами, в процессе вычисления результата СУБД исходит из предположения, что данные в таблицах-источниках соответствуют моменту старта выполнения SELECT, сколько бы это выполнение не длилось. (Хотя это явно никак не выражено, можно считать, что на время вычисления СУБД организует в пределах сеанса «автономную» транзакцию типа READ ONLY).

Иллюзия неизменности данных таблиц-источников в процессе вычисления запроса требует от СУБД расходования внутренних ресурсов. Если запрос длится особо долго, а данные в это время активно изменяются прочими сеансами, требование логической целостности может вступить в конфликт с ресурсными ограничениями СУБД, и тогда обработка оборвется, а программа вместо ответа получит сообщение об ошибке. Такие проблемы решаются настройкой приложения или СУБД программистом и администратором.

6.3. Фраза FROM предложения SELECT

Фраза FROM — одна из двух обязательных для каждого предложения SELECT. Она перечисляет источники данных для получения ответа на запрос. Логически она открывает цепочку вычислений, осуществляя (логически !) декартово произведение строк всех источников друг с другом.

Если в предложении SELECT есть подзапросы, фраза FROM основного запроса будет не единственной, где в конечном итоге указываются источники данных, однако в подзапросах такие источники все равно будут указываться в собственных фразах FROM.

6.3.1. Варианты указания таблицы-источника во фразе FROM и поля строки в остальных фразах

Безотносительно к форме ссылки на источник (краткое или расширенное имя таблицы), сослаться на его столбцы можно как по кратким (если только не возникает двусмыслицы), так и по уточненным именам:

```
SELECT ename, emp.ename FROM emp;  
-- Допустимо.
```

```
SELECT ename, emp.ename, scott.emp.ename FROM scott.emp;  
-- Допустимо.
```

```
SELECT scott.emp.ename FROM emp;  
-- Не допустимо, ошибка !
```

Полностью уточненные имена в запросе удлиняют формулировку, но (а) снимают неоднозначность понимания программистом или же смысла системой, и (б) дают некоторое ускорение обработки запроса, так как Oracle все равно будет пытаться определить расширенное имя.

6.3.2. Использование псевдонимов в запросе

«Псевдоним» — это такой синоним таблицы, действие которого ограничено рамками текста конкретного запроса. Он может служить для придания запросу лаконичности, читаемости или же гибкости формулировки.

Если во фразе FROM таблице приписан псевдоним, то во всех остальных фразах запроса SQL требует ссылаться на нее только по псевдониму.

6.3.2.1. Пример

Пример использования для краткости формулировки:

```
SELECT ename, dname  
FROM   emp e, dept d  
WHERE  e.deptno = d.deptno  
;
```

Выше этот же запрос приводился без псевдонимов, с указанием фактических имен таблиц во фразе WHERE.

Пример использования для повышения ясности текста:

```
SELECT employee.ename, department.dname  
FROM   emp employee, dept department  
WHERE  employee.deptno = department.deptno  
;
```

6.3.2.2. Когда псевдонимы обязательны

Есть случаи, когда использование синонимов представляет из себя не вопрос удобства, а меру необходимости.

«Выдать сотрудников, принятых на работу позже президента»:

```
SELECT employees.ename  
FROM   emp employees, emp topmanagers  
WHERE  topmanagers.job = 'PRESIDENT'  
AND    employees.hiredate > topmanagers.hiredate  
;
```

Использование во фразе FROM для ссылки на источник несколько раз одной и той же таблицы (выступающей содержательно в разных качествах; здесь — как перечень руководящих лиц и перечень сотрудников) требует употребления псевдонимов. Они-то и позволят сослаться в выражениях на поля строк таблицы в нужном смысле.

6.3.3. Подзапрос в качестве источника данных

Начиная с версии 8 во фразе FROM в качестве источника данных вместо таблицы может быть указан вложенный запрос:

```
SELECT  ename, dname
FROM    emp e, ( SELECT * FROM dept WHERE loc <> 'NEW YORK' ) d
WHERE   e.deptno = d.deptno
;
```

Чтобы в выражениях сослаться на поля строк результата подзапроса (что на практике очень вероятно), подзапрос обязан наделаться псевдонимом.

Подзапрос в качестве источника данных разрешен не только в предложении SELECT, но и в INSERT, UPDATE и DELETE.

6.3.4. Указание для таблицы пробной выборки строк

Обычно указанный во фразе FROM источник поставляет для последующей обработки полное множество своих строк. Версия 9 позволила указать для таблицы-источника неполную выборку строк. В этом случае в работу поступят не все строки, а случайная их выборка. В стандарте SQL:2003 подобная возможность указывается через уточнение TABLESAMPLE, синтаксически несколько отклоняющееся от предлагаемого Oracle.

Ниже приводится несколько примеров способов указать такую выборку.

Отбор примерно 10% всех имеющихся сотрудников:

```
SELECT  ename, sal FROM emp SAMPLE ( 10 );
SELECT COUNT ( * ) FROM emp SAMPLE ( 10 );
```

Упражнение. Проверить, что оба запроса выше будут возвращать новые и новые результаты (на деле они будут заметно повторяться, но только в силу небольшого размера таблицы EMP).

Внутренняя техника основывается на применении для отбора строк псевдослучайной последовательности чисел. Поэтому обычно повторение такого запроса будет выдавать все разные и разные строки.

Если по каким-то причинам желательно выборку зафиксировать, и сделать одинаковой при повторениях запроса (допустим, неполную выборку строк желательно воспроизводить без изменений), в текст запроса добавляется конкретная «затравка» (seed) для вычисления внутренней псевдослучайной последовательности, например:

```
SELECT  ename, sal FROM emp SAMPLE ( 10 ) SEED ( 3 );
```

Другой способ задать (псевдо-)случайную выборку строк состоит в указании общего процента данных, отбираемых для обработки из каждого блока со строками таблицы в БД:

```
SELECT COUNT ( * ) FROM emp SAMPLE BLOCK ( 85 );
```

В качестве источника может выступать не обязательно фактическая таблица, но и представление данных (view; см. ниже), однако когда его формулировка не содержит обобщений. Частичную выборку не запрещено указать и в многотабличных запросах:

```
SELECT  ename, dname
FROM    emp SAMPLE ( 10 ), dept
WHERE   emp.deptno = dept.deptno
;
```

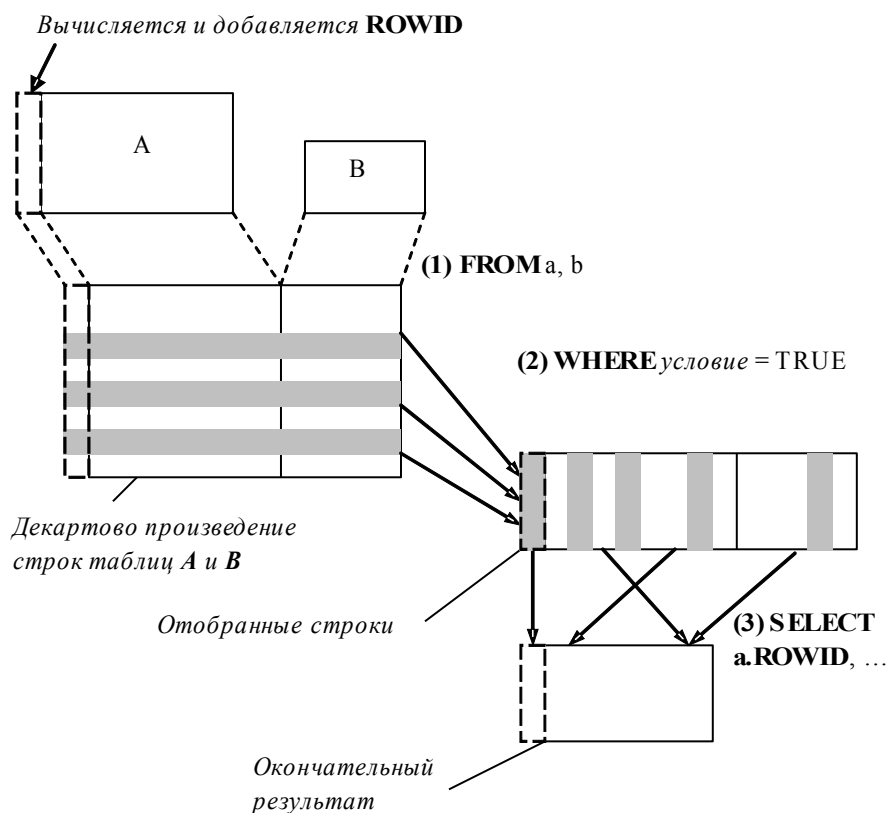
Указание частичной выборки похоже на выборку с отсевом строк `WHERE ROWNUM <= n` (см. ниже) при том, что последняя и менее трудоемка. Однако при употреблении условия с `ROWNUM` речь идет об отборе произвольных *n* строк («квотированной выборке»), а в запросе с `SAMPLE` — о примерной доле общего количества. Вдобавок, отбор с `SAMPLE` в состоянии дать более представительную выборку строк.

6.3.5. Специальный синтаксис для запроса-соединения

Если предложение `SELECT` представляет из себя особый случай соединения (`join`) источников данных, то во фразе `FROM` могут использоваться другие синтаксические построения. Их употребление раскрыто далее в разделе, относящемся к этой операции.

6.3.6. Физический адрес ROWID строки в таблице

Если во фразе `FROM` в качестве источника данных приведена хранимая таблица, то на этапе извлечения ее строк СУБД определяет для них физические адреса. Они становятся доступны в остальных частях запроса посредством «псевдостолбца» (на деле — функции без параметров) `ROWID`. Например, логическая схема обработки предложения вида `SELECT a.ROWID, ... FROM a, b WHERE условие` может выглядеть так:



Физический адрес `ROWID` представлен типом `ROWID`, под которым может сохраняться в БД. Он содержит четыре компонента: номер объекта (используется для таблицы из объектов), номер файла, номер блока и номер строки в блоке. Он расходует 10 байтов, но `SQL*Plus` выдает его значение в формате `BASE64` 18-ю символами:

```
SQL> COLUMN dump(rowid) FORMAT A40
```

```
SQL> SELECT ROWID, DUMP ( ROWID ), loc FROM dept;
```

ROWID	DUMP (ROWID)	LOC
AAAR3qAAEAAAACHAAA	Typ=69 Len=10: 0,1,29,234,1,0,0,135,0,0	NEW YORK
AAAR3qAAEAAAACHAAB	Typ=69 Len=10: 0,1,29,234,1,0,0,135,0,1	DALLAS
AAAR3qAAEAAAACHAAC	Typ=69 Len=10: 0,1,29,234,1,0,0,135,0,2	CHICAGO
AAAR3qAAEAAAACHAAD	Typ=69 Len=10: 0,1,29,234,1,0,0,135,0,3	BOSTON

Хранение физического адреса в БД должно быть исключительной мерой, так как в целом он ненадежен. В результате изменений в БД одно и то же значение ROWID может со временем указывать на совсем иную строку (объект) даже в другой таблице, чем это было первоначально. При этом ROWID находит себе место в некоторых технических запросах, например, в одном из способов устранить из таблицы повторяющиеся строки. Также нередко программисты увлекаются тем обстоятельством, что выборка строки по условию WHERE ROWID = *адрес* является в Oracle самой быстрой.

Еще реже находит себе применение другой «псевдостолбец» — UROWID, имеющий аналогичный смысл и вычисляемый при извлечении строк из таблиц с индексной организацией хранения и с внешним хранением.

6.4. Фраза WHERE предложения SELECT

*...Явились тут на нескольких листах;
 Какрой-то Шмидт, два брата Шулаковы,
 Зерцалов, Палкин, Савич, Розенбах,

 Мадам Гриневиц, Глазов, Рыбин, Штих,
 Бурдюк-Лишай — и множество других;
 А. К. Толстой. Сон Попова*

Отточие в цитате из русского классика имеет тот же смысл, что и в приводившихся ранее ответах Oracle: рамки технического документа стеснительны для хорошей поэзии, равным образом как и для хорошей информационной системы. Чтобы сделать увлекательный, но несколько затянутый список более обозримым, Oracle рекомендует применить к стихотворению графа Толстого фразу WHERE.

Фраза WHERE принимает на входе строки, полученные в результате вычисления предшествующей фразы, отбирает из этих строк те, что дают значение «истина» для некоторого условного выражения, и отобранные таким образом строки передает следующим фразам в общей схеме выполнения запроса. Выполняемое ею действие унаследовано из реляционной модели, от операции, называемой там «ограничением» (выбраковкой, restriction). По этой причине, собственно ей бы и следовало дать в SQL обозначение SELECT (от латинского selectus — «выбранный»), но создатели SQL отнесли это обозначение на счет данных вообще, а не строк.

Логический алгоритм отработки фразы WHERE прост:

```

Результат WHERE := пусто;
для каждой Строки во Входном множестве выполнить:
    если Условное выражение = TRUE то
        Результат WHERE :=+ Строка;
конец цикла;
```

Ниже говорится о способах формирования условных выражений ради отбора строк фразой WHERE. Они строятся из обычных выражений с помощью операций сравнения и ряда иных, а также из прочих условных выражений с помощью логических операций.

6.4.1. Операторы сравнения значений

Сравнение «обычных» (числовых, строковых или временных) значений друг с другом — традиционный и очевидный способ получить условное выражение:

<i>Оператор сравнения</i>	<i>Действие</i>
=	равно
<	меньше
>	больше
<=	меньше или равно
>=	больше или равно
<>, !=, ^=	не равно

В силу исторических причин сравнение на неравенство имеет три равносильных обозначения. В стандарт SQL из них входит только <>.

Объектные типы сравнивать можно, только если это предусмотрено программистом БД в определении типа. Данные встроенных объектных типов XMLTYPE, или ANYDATA, например, сравнивать нельзя.

6.4.1.1. Сравнение строк текста

Строки типа VARCHAR2 при сравнении могут иметь разную длину. Сравнение происходит посимвольно слева направо, покуда одна из строк не закончится. Примеры для SQL*Plus:

```
VARIABLE v2 VARCHAR2 ( 2 )
VARIABLE v3 VARCHAR2 ( 3 )
SELECT
  CASE WHEN :v2 = :v3 THEN '=' WHEN :v2 < :v3 THEN '<' END
FROM dual
.
```

SAVE compare REPLACE

Далее:

```
SQL> EXECUTE :v2 := 'ab'; :v3 := 'ab'

PL/SQL procedure successfully completed.

SQL> @compare

C
-
=

SQL> EXECUTE :v2 := 'ab'; :v3 := 'ab '

PL/SQL procedure successfully completed.

SQL> @compare

C
-
<

SQL> EXECUTE :v2 := 'ab'; :v3 := 'ab' || chr ( 0 )

PL/SQL procedure successfully completed.
```

```
SQL> @compare
```

```
C
-
<
```

Строки текста типа CHAR имеют в Oracle при сравнении неочевидную особенность, уходящую корнями в стандарт SQL. Она состоит в игнорировании хвостовых пробелов при сравнении (на равенство или неравенство — не важно), например:

```
SQL> SELECT 'ok' FROM dual WHERE 'Париж' = 'Париж ';
```

```
'O
--
ok
```

```
SQL> SELECT 'ok' FROM dual WHERE 'Париж' <> 'Париж ';
```

```
no rows selected
```

Одновременно, эти «одинаковые» строки имеют в SQL все-таки разные свойства:

```
SQL> SELECT LENGTH ( 'Париж' ), LENGTH ( 'Париж ' ) FROM dual;
```

```
LENGTH ( 'ПАРИЖ' )  LENGTH ( 'ПАРИЖ' )
-----
                    5                6
```

При выполнении операций обобщения или устранения дубликатов по столбцу (например, с использованием DISTINCT, GROUP BY или UNION), внутренняя логика осуществления которых подразумевает сравнение, такое поведение, по замечанию Кристофера Дейта, способно довести прикладного программиста до состояния медитации, официально делая результат запроса к БД непредсказуемым. Опыты показывают, что СУБД Oracle все же отошла здесь от стандарта и во *внутренних* сравнениях полагает, что 'Париж' = 'Париж ' не дает TRUE. Справедливости ради можно обратить внимание, что именно перечисленные операции типично будут выполнять внутренние сравнения на значениях, собранных в столбец, и будучи при этом приведенных СУБД к общему формату (попросту говоря — дополненных нужным количеством пробелов), вследствие чего проблема себя не проявит.

6.4.2. Логические операторы AND, OR и NOT в логических выражениях

Подобно выражениям прочих типов, логические могут строиться на основе более простых с помощью собственных, логических операторов. Таковыми служат бинарные AND и OR, а также унарный NOT. Отсутствие величины, обозначаемое как NULL, для логических значений допускается, как и для прочих. Формально символ NULL может восприниматься здесь как третье допустимое значение, дополняющее TRUE и FALSE, а это приводит SQL к трехзначной логике.

В трехзначной логике таблицы значений для логических операторов выглядят следующим образом:

AND:

	TRUE	NULL	FALSE
TRUE	TRUE	NULL	FALSE
NULL	NULL	NULL	FALSE
FALSE	FALSE	FALSE	FALSE

OR:

	TRUE	NULL	FALSE
TRUE	TRUE	TRUE	TRUE
NULL	TRUE	NULL	NULL
FALSE	TRUE	NULL	FALSE

NOT:

<i>NOT (TRUE)</i>	<i>NOT (NULL)</i>	<i>NOT (FALSE)</i>
FALSE	NULL	TRUE

Встроенная шкала приоритетности выполнения операций допускает однозначные бесскобочные формулировки. Например, условие $C_1 \text{ OR } C_2 \text{ AND } C_3$ обрабатывается так:

- (1) $C_2 \text{ AND } C_3 \rightarrow a_1$
- (2) $C_1 \text{ OR } a_1 \rightarrow \text{результат}$

Тем не менее методически бесскобочной формулировкой рекомендуется пользоваться с осторожностью, только в самых бесспорных случаях. Чрезмерное ее применение способствует ошибкам программирования, равно как и расстановка очевидных скобок без особой нужды.

Упражнение. Выдать всех сотрудников, кроме клерков и продавцов.

Следует обратить внимание на асиметрию отношения операций AND и OR к NULL в качестве одного из операндов. Содержательно она соответствует пониманию NULL как «неизвестно чего» (unknown): то ли TRUE, то ли FALSE, но чего-то из двух (а в стандарте SQL:1999 для булевского типа «третье» значение носит официальное название UNKNOWN). К сожалению, SQL ничего не противопоставляет иному пониманию NULL, в жизни допустимому (SQL как таковой вообще никак не учитывает никакое «понимание» и не связан с ним). Если в конкретных обстоятельствах NULL содержательно означает «значение отсутствует», булева логика SQL должна для программиста превратиться в чистую формальность, никак не связанную с каким бы то ни было «интуитивным пониманием».

Сравните два примера замены прямого условного выражения на обратное:

```
SQL> SELECT 'ok' FROM dual WHERE 1 = 1 OR 1 = NULL;

'О
--
ok

SQL> SELECT 'ok' FROM dual WHERE NOT ( 1 = 1 OR 1 = NULL );

no rows selected
```

В то же время:

```
SQL> SELECT 'ok' FROM dual WHERE 1 = 2 OR 1 = NULL;

no rows selected

SQL> SELECT 'ok' FROM dual WHERE NOT ( 1 = 2 OR 1 = NULL );

no rows selected
```

Замена прямого условия на обратное требует внимания программиста: как минимум, в SQL она не подвержена двузначной логике с психологически понятным человеку правилом «или [одно] — или [другое]», «третьего не дано».

6.4.3. Вычисление составного логического выражения

Здесь Люба [...] не набралась больших знаний, но [...] все-таки узнала гораздо более, чем считал для нее нужным и полезным
Павлин. "

Н. С. Лесков. Павлин

При использовании цепочек из AND и OR логические выражения вычисляются не обязательно полностью, а до установления ясности, каким будет результат независимо от дальнейшего вычисления подвыражений. Иногда это возможно. Oracle называет это short-circuit evaluation, или «короткое вычисление» выражений. Цепочки вычисляются обыкновенно справа налево (AND) или слева направо (OR). Однако на деле не все так прямолинейно, и на эту технику может наложиться выявление тривиальных условных подвыражений, выполняемое на подготовительной фазе разбора. Все это вместе способствует затаенным ошибкам, угадать появление которых при программировании запросов не всегда просто. Сравните две пары запросов:

```
SQL> SELECT 'ok' FROM dual WHERE 1 = 1 / 0 OR 1 = 1;
```

```
'O
--
ok
```

```
SQL> SELECT 'ok' FROM dual WHERE 1 = 1 OR 1 = 1 / 0;
```

```
'O
--
ok
```

```
SQL> SELECT 'ok' FROM dual WHERE 1 = ( SELECT 1 FROM dual ) OR 1 = 1 / 0;
```

```
'O
--
ok
```

```
SQL> SELECT 'ok' FROM dual WHERE 1 = 1 / 0 OR 1 = ( SELECT 1 FROM dual );
SELECT 'ok' FROM dual WHERE 1 = 1 / 0 OR 1 = ( SELECT 1 FROM dual )
*
```

```
ERROR at line 1:
ORA-01476: divisor is equal to zero
```

Очевидно, оптимизатор, разбирая условное выражение, сначала определяет тривиальные подвыражения чисто аналитически (еще не приступая к фактической оценке). Хотя и не часто, но таковые бывают. В нашем случае это $1 = 1$. Обнаружив в структуре логического выражения цепочку из AND или OR, оптимизатор чисто формально проверит, достаточно ли выявленных значений тривиальных подвыражений для вынесения окончательного решения по поводу результата. И только если недостаточно, начинается фактическое оценивание подвыражений. При прочих равных порядок оценивания — слева направо или справа налево, но на деле в него могут дополнительно вмешаться: более раннее вычисление подзапросов; обращения к встроенным функциям, для которых СУБД известна стоимость вычислений; имеющаяся для таблиц — объектов доступа статистика хранения; до версии 10 — подсказка ORDERED_PREDICATES оптимизатору. Вот несколько поясняющих примеров:

```
SELECT 'ok' FROM dual WHERE 1 = 1 / 0 OR 1 = TRUNC ( 1 );
-- ok
SELECT 'ok' FROM dual WHERE 1 = TRUNC ( 1 ) OR 1 = 1 / 0;
-- ok
SELECT 'ok' FROM dept WHERE 1 = 1 / 0 OR LENGTH ( loc ) = LENGTH ( loc );
-- ok
SELECT 'ok' FROM dept WHERE LENGTH ( loc ) = LENGTH ( loc ) OR 1 = 1 / 0;
-- ok
SELECT 'ok' FROM dept WHERE 1 = 1 / 0 OR 1 = LENGTH ( loc ) / LENGTH ( loc );
-- ORA-01476: divisor is equal to zero
SELECT 'ok' FROM dept WHERE 1 = LENGTH ( loc ) / LENGTH ( loc ) OR 1 = 1 / 0;
-- ok
```

Такая техника оценки логического выражения способна экономить вычисления. Этим обстоятельством может пользоваться программист, размещая наиболее достоверные или легковычисляемые подвыражения в соответствующем краю цепочки (особенно когда в подвыражениях встречаются обращения к функциям пользователя и СУБД вынуждена применять стандартный порядок). Об оборотной стороне уже упоминалось: такая техника вычислений снижает предсказуемость содержательной корректности общего выражения при построении последнего программистом.

6.4.4. Условный оператор IS

Типы данных в Oracle и в стандарте SQL достаточно разнообразны, так что традиционный оператор сравнения на равенство = не всегда в состоянии их адекватно обслуживать. Для сравнения на равенство в ряде «нестандартных» случаев используется особый оператор IS.

При этом, наиболее распространенное его употребление — в проверке на отсутствие значения-результата после вычисления выражения.

6.4.4.1. Сравнение IS NULL

Рассмотрим запрос: «Выдать сотрудников, не имеющих комиссионного вознаграждения, и их оклады». Следующие две попытки очевидно показывают ошибочность формулировки условного выражения со сравнением в конкретном случае:

```
SQL> SELECT ename, sal FROM emp WHERE comm = NULL;
```

no rows selected

```
SQL> SELECT ename, sal FROM emp WHERE comm <> NULL;
```

no rows selected

Но этого и следовало ожидать: общее правило SQL гласит, что обычное сравнение значений (в том числе операциями = и <>) даст NULL, если один из операндов NULL. Логика в том, что при сравнении «неизвестно чего» и результат «неизвестно какой». Приводившийся выше алгоритм работы фразы WHERE сообщает, что если условное выражение оценивается как NULL, строка в дальнейшую обработку не отбирается. По этой причине сравнения = NULL и <> NULL во фразе WHERE (равно как во всех других допустимых местах) бессмысленны, так как всегда приведут к пустому результату.

Для выявления наличия или отсутствия значений используются особые операторы IS NULL и IS NOT NULL:

```
SQL> SELECT ename, sal FROM emp WHERE comm IS NULL;
```

ENAME	SAL
SMITH	800
JONES	2975
BLAKE	2850
CLARK	2450
SCOTT	3000
KING	5000
ADAMS	1100
JAMES	950
FORD	3000
MILLER	1300

Выражение:

E_1 IS NOT NULL

равносильно:

NOT (E_1 IS NULL)

6.4.4.2. Особенности проверки на отсутствие значения для типа VARCHAR2

В противоречие стандарту ANSI/ISO Oracle считает строку без символов *отсутствующей* строкой типа VARCHAR2:

```
SQL> VARIABLE nochars VARCHAR2 ( 10 )
```

```
SQL> SELECT 'ok' FROM dual WHERE :nochars IS NULL;
```

```
'O  
--  
ok
```

```
SQL> SELECT 'ok' FROM dual WHERE :nochars = '';
```

```
no rows selected
```

Сказанное распространяется и на NVARCHAR2.

Для типа CLOB (равно как и NCLOB) различие между пустой строкой и строкой без символов, как и в стандарте, проводится, однако сам он устроен сложнее. По стандарту для типов LOB существует локатор (который хранится вместе с другими полями строки в БД) и отдельно хранимый массив байтов. В условных выражениях SQL типы LOB допускают сравнения как раз только операторами IS NULL/IS NOT NULL. Прочие сравнения устраиваются посредством функций из пакета DBMS_LOB. В PL/SQL таких ограничений на условные выражения с типами LOB нет.

6.4.4.3. Сравнения для числовых данных BINARY_FLOAT и BINARY_DOUBLE

Стандарт IEEE 754 предписывает для данных этих типов возможность специальных значений Not a Number и +/- Infinity. Примеры сравнений в Oracle:

```
SELECT 'ok' FROM dual WHERE 123f IS NOT NAN;
```

```
SELECT 'ok' FROM dual WHERE TO_BINARY_DOUBLE ( '-INF' ) IS INFINITE;
```

```
SELECT 'ok' FROM dual WHERE BINARY_FLOAT_INFINITY - 1 IS INFINITE;
```

6.4.4.4. Сравнения для объектных данных

Оператор IS используется также для составления условных выражений с участием объектных данных Oracle. Вот примеры, как это могло бы выглядеть в некоторой гипотетической базе данных:

```
SELECT * FROM checkpoint WHERE person IS OF ( employee_type );  
-- Отбор объектов типа EMPLOYEE_TYPE
```

```
SELECT dname FROM dept WHERE addr IS DANGLING;  
-- Отбор строк с ссылками на более не существующие в БД объекты
```

Для «вложенных таблиц» (элемент объектных возможностей Oracle) сравнения могли бы выглядеть так:

```
SELECT model_type FROM colour_models WHERE colours IS EMPTY;  
-- Названия цветовых моделей с пустым множеством их элементов
```

```
SELECT model_type FROM colour_models WHERE colours IS A SET;  
-- Названия цветовых моделей с отсутствием повторений среди их элементов
```

6.4.4.5. Сравнения для данных JSON

С версии 12 — проверка на соответствие формату JSON, по типу:

```
SELECT 'notJSON' FROM dual WHERE dummy IS NOT JSON;
```

6.4.5. Условный оператор LIKE

Оператор LIKE рассчитан на выявление наличия в строке текста подстроки, задаваемой в виде шаблона. В определении шаблона используются два специальных символа:

- `_` — один любой символ;
- `%` — отсутствие символа или цепочка любых символов.

Примеры:

```
SELECT ename FROM emp WHERE ename LIKE 'M%';
-- сотрудники, имена которых начинаются на 'M' (а возможно, этим и оканчиваются)

SELECT ename FROM emp WHERE ename LIKE '%AR_';
-- сотрудники, две предпоследних буквы имени которых — 'AR'

SELECT ename, sal FROM emp WHERE sal LIKE 2 || '%';
-- автоматическое преобразование типов
```

Когда необходимо, чтобы символ `_` или `%` в шаблоне играл обычную, а не особую роль, его следует предварить каким-нибудь произвольно выбранным символом, о котором сообщить отдельно в продолжении `ESCAPE` оператора `LIKE`. Например, требуется найти таблицы словаря-справочника, выдающие те или иные сведения о столбцах таблиц в моей схеме. Из опыта работы со словарем-справочником Oracle программист знает, что имена подобных таблиц скорее всего содержат шаблон `'_COL_'`. Правильно составленный запрос будет таким:

```
COLUMN comments FORMAT A60 WORD

SELECT table_name, comments
FROM dictionary
WHERE table_name LIKE 'USER%\_COL\_%' ESCAPE '\';
```

Упражнение. Сравнить ответ на предыдущий запрос с ответом на другой, в котором шаблон сравнения указан просто как `'USER%_COL_'`.

Для отрицания пишется `NOT LIKE`, например:

```
SELECT ename FROM emp WHERE ename NOT LIKE 'M%';
```

Начиная с версии 9.2 оператор `LIKE` (вместе с некоторыми строковыми функциями, например, `INSTR`, `SUBSTR`) применим не только к типам `CHAR` и `VARCHAR2`, но также к `CLOB` и `NCLOB`.

Помимо `LIKE`, имеются еще операторы `LIKEC`, `LIKE2` и `LIKE4`, реализующие разные логики сравнения символов в Unicode.

6.4.6. Условный оператор REGEXP_LIKE

Выразительные возможности оператора `LIKE` невелики. С версии 10 в Oracle существует намного более мощный оператор `REGEXP_LIKE`, позволяющий делать проверку наличия в тексте шаблона, построенного техникой регулярных выражений. Пример:

```
SELECT ename FROM emp WHERE REGEXP_LIKE ( ename, ' (^\.LL|TT)' );
-- сотрудники, с двумя подряд 'T' где угодно или с двумя подряд 'L' на втором и
-- третьем местах сначала
```

Как видно, формально оператор `REGEXP_LIKE` устроен традиционно, а не инфиксно, подобно `LIKE`. Однако соответствующий ему в SQL:1999 оператор `SIMILAR` устроен инфиксно.

Другой пример. Требуется найти таблицы словаря-справочника, выдающие те или иные сведения о столбцах таблиц, как в моей схеме, так и в чужих, когда те мне доступны. Запрос можно построить так:

```
SELECT table_name
FROM dictionary
WHERE REGEXP_LIKE ( table_name, '^ (USER|ALL) .* _COL_ ' )
;
```

Возможный вариант того же запроса, когда регистр не играет роли (а может быть, заранее не известен):

```
SELECT table_name
FROM dictionary
WHERE REGEXP_LIKE ( table_name, '^ (user|all) _col_ ', 'i' )
;
```

В третьем аргументе REGEXP_LIKE значение 'i' означает case-insensitive, то есть неразличение верхнего и нижнего регистра.

Регулярные выражения используются, кроме того, для анализа и изменения строк функциями REGEXP_COUNT, REGEXP_INSTR, REGEXP_REPLACE и REGEXP_SUBSTR, доступными в числовых и текстовых выражениях SQL. Например, в приложении для web можно осуществлять программную подмену адресов URL:

```
SELECT REGEXP_REPLACE (
    'persinfo?p=123'
    , ' ("persinfo") (\?) (p) (=) ([[digit:]]+) (")'
    , '\1\3\5.html\6'
)
FROM dual
;
```

В данном случае "persinfo?p=123" → "persinfo123", но число можно указывать любое целое положительное.

Для составления регулярных выражений Oracle дает достаточно широкий набор специальных символов и условных обозначений, полный перечень которых имеется в документации. В целом правила составления таких выражений в Oracle напоминают принятые в Perl, .NET, Java и установленные стандартом POSIX, но содержат и отличия от всего перечисленного.

6.4.7. Условный оператор BETWEEN

Используется для проверки «попадания» значения в указанный диапазон. Пример запроса:

```
SELECT ename FROM emp WHERE sal BETWEEN 1000 AND 2000;
```

В общем случае все операнды оператора BETWEEN представляют из себя выражения.

Выражение:

$E_1 \text{ BETWEEN } E_2 \text{ AND } E_3$

равносильно;

$(E_1 \geq E_2) \text{ AND } (E_1 \leq E_3)$

Следствия

- 1) Оператор действует на значениях всех типов, допускающих сравнения на неравенство (числовых, символьных, временных, и в некоторых случаях объектных).
- 2) Если хотя бы одно из E_n будет NULL, результат будет NULL.

Выражение:

$E_1 \text{ NOT BETWEEN } E_2 \text{ AND } E_3$

равносильно:

$\text{NOT} (E_1 \text{ BETWEEN } E_2 \text{ AND } E_3)$

и равносильно:

$(E_1 < E_2) \text{ OR } (E_1 > E_3)$

По причине наличия равносильных более общих формулировок, оператор BETWEEN ничего содержательно нового в SQL не привносит. Его ценность в ином:

- он способствует лучшему пониманию текста программистом, что понижает шанс для ошибок;
- он вычисляется эффективнее своей более традиционной равносильной переформулировке.

Использование BETWEEN в запросах можно всячески приветствовать, но только если у его трех операндов значения присутствуют. Отсутствия значений (NULL) у операндов может ввести программиста в заблуждение относительно результата будущего ответа, если только он не сумеет, формулируя запрос, переключиться со своей человеческой логики на формальную логику SQL. Это не всегда психологически легко, и поэтому требует от программиста особого внимания. Пример:

```
SQL> SELECT ename, comm FROM emp WHERE comm BETWEEN 10 AND 1000;
```

ENAME	COMM
ALLEN	300
WARD	500

```
SQL> SELECT ename, comm
2 FROM emp
3 WHERE comm BETWEEN ( SELECT comm FROM emp WHERE ename = 'SCOTT' )
4 AND 1000
5 ;
```

no rows selected

Первый запрос выдан для противопоставления второму. Обратите внимание: во втором случае мы *не* получили в результате даже сведений о самом сотруднике SCOTT.

6.4.8. Условный оператор IN с явно перечисляемым множеством

Используется для проверки совпадения значения с одним из перечисленных явно. Пример:

```
SELECT ename FROM emp WHERE job IN ( 'MANAGER', 'ANALYST' );
```

В общем случае все участники такого сравнения могут быть составными выражениями, а не обязательно явными значениями.

Выражение:

$E_1 \text{ IN } (E_2, E_3, E_4, \dots)$

равносильно:

$(E_1 = E_2) \text{ OR } (E_1 = E_3) \text{ OR } (E_1 = E_4) \text{ OR } \dots$

Следствия

- 1) Оператор действует на значениях всех типов, допускающих сравнения на равенство (числовых, символьных, временных, в некоторых случаях объектных).
- 2) Если E_1 будет NULL, результат будет NULL.

Оператор IN технологически подобен BETWEEN: он ничего нового в SQL содержательно не привносит, имеет те же общие выгоды перед равносильной более традиционной формулировкой и равным образом требует осторожности при обращении с отсутствующими значениями.

Выражение:

$E_1 \text{ NOT IN } (E_2, E_3, E_4, \dots)$

равносильно:

$\text{NOT } (E_1 \text{ IN } (E_2, E_3, E_4, \dots))$

равносильно:

$(E_1 \lessdot E_2) \text{ AND } (E_1 \lessdot E_3) \text{ AND } (E_1 \lessdot E_4) \text{ AND } \dots$

Элементами сравнения могут выступать списки значений:

```
SELECT ename FROM emp
WHERE ( job, sal ) IN ( ( 'ANALYST', 3000 ), ( 'MANAGER', sal ) )
;
```

(Аналитики с зарплатами 3000 и все менеджеры, получающие зарплату).

Еще пример:

```
SELECT 'ok' FROM dual
WHERE ( 1, 2, 3 ) IN ( ( 1, 2, 3 ), ( 5, 6, 7 ) )
;
```

6.4.9. Условный оператор IN с множеством, получаемым из БД

Вторая разновидность оператора IN позволяет установить равенство с хотя бы одним элементом множества, извлекаемого подзапросом из БД, а не перечисляемым явно.

«Выдать сотрудников, работающих в Нью-Йорке или Бостоне»:

```
SELECT ename FROM emp WHERE deptno IN ( 10, 40 );
```

Однако если мы не помним номера отделов Нью-Йорке и Бостоне, запрос естественно сформулировать иначе:

```
SELECT ename
FROM emp
WHERE deptno IN ( SELECT deptno
                  FROM dept
                  WHERE loc IN ( 'NEW YORK', 'BOSTON' )
                  )
;
```

«Выдать сотрудников, имеющих те же должности и руководство, что у работающих в Далласе»:

```
SELECT ename
FROM emp
WHERE ( job, mgr ) IN ( SELECT job, mgr
                      FROM emp
                      WHERE deptno IN ( SELECT deptno
                                      FROM dept
                                      WHERE loc = 'DALLAS'
                                      )
                      )
;
```

Выражение:

$C \text{ IN } (S)$

равносильно:

$(C = v_1) \text{ OR } (C = v_2) \text{ OR } \dots \text{ OR } (C = v_n) \text{ OR FALSE}$

Следствия

- 1) Если подзапрос для множества значений не выдает результата, условие отбора становится FALSE.
- 2) Если C будет NULL, а подзапрос возвращает хотя бы одно значение, результат будет NULL.

Выражение:

$C \text{ NOT IN } (S)$

равносильно:

$(C \neq v_1) \text{ AND } (C \neq v_2) \text{ AND } \dots \text{ AND } (C \neq v_n) \text{ AND TRUE}$

Упражнение. Как можно сформулировать последствия наличия такой равносильной формулировки ? (См. также пример в конце материалов).

6.4.10. Кванторы ANY и ALL при сравнении с элементами множества значений

Сравнение в условном выражении значения с множеством значений синтаксически невозможно, однако постановка кванторов ANY и ALL перед указанием множества такую возможность открывает. Как и в случае с оператором IN, множество может быть перечислено явно, а может выбираться из БД подзапросом; точно так же допустимо сравнение списков, а не отдельных значений.

Квантор ALL в операторе сравнения приведет к истинному результату, если значение справа от знака сравнения истинно сравнивается *со всеми* элементами множества. Примеры:

«Выдать сотрудников, у которых зарплата не 1000 и не 1500»:

```
SELECT ename
FROM    emp
WHERE   sal <> ALL ( 1000, 1500 )
;
```

Все участники такого сравнения, как и в случае с оператором IN, могут быть составными выражениями.

«Выдать самых старых сотрудников»:

```
SELECT ename
FROM    emp
WHERE   hiredate <= ALL ( SELECT hiredate FROM emp )
;
```

Выражение:

$E \leq \text{ALL} (S)$

равносильно:

$(E \leq v_1) \text{ AND } (E \leq v_2) \text{ AND } \dots \text{ AND } (E \leq v_n) \text{ AND TRUE}$

Упражнение. Что будет:

- в результате, если подзапрос не вернет данных ?
- если одно из v_i будет NULL ?
- если E будет NULL ?

Квантор ANY в операторе сравнения приведет к истинному результату, если значение справа от знака сравнения истинно сравнивается *хотя бы с одним* элементом множества. Примеры:

«Выдать сотрудников, у которых зарплата 1000 или 1500»:

```
SELECT ename
FROM    emp
WHERE   sal = ANY ( 1000, 1500 )
;
```

«Выдать сотрудников, нанятых позже кого-нибудь из отдела 20»:

```
SELECT ename
FROM    emp
WHERE   hiredate > ANY ( SELECT hiredate FROM emp WHERE deptno = 20 )
;
```

(Выдаст частью и самих сотрудников отдела 20).

Выражение:

$E > \text{ANY} (S)$

равносильно:

$(E > v_1) \text{ OR } (E > v_2) \text{ OR } \dots \text{ OR } (E > v_n) \text{ OR FALSE}$

Упражнение. Сформулировать последствия наличия подобной равносильной формулировки.

Сравнения с применением кванторов ANY и ALL добавляют программисту удобств в формулировании условных выражений, но теоретически приносят в SQL избыточность. Обратите внимание, что $= \text{ANY}$ равносильно IN , а $\neq \text{ALL}$ равносильно NOT IN , но иногда один из этих двух видов оформления лучше передает смысл запроса. Сравните, например, две равносильные (с точностью до плана выполнения) формулировки запроса на выдачу наиболее высокооплачиваемых сотрудников в каждом отделе:

```
SELECT ename, sal, deptno
FROM    emp
WHERE   ( sal, deptno ) IN ( SELECT  MAX ( sal ), deptno
                             FROM    emp
                             GROUP BY deptno )
;
```

```
SELECT ename, sal, deptno
FROM    emp
WHERE   ( sal, deptno ) = ANY ( SELECT  MAX ( sal ), deptno
                             FROM    emp
                             GROUP BY deptno )
;
```

Для лучшего восприятия текста запроса программистом, для которого английский язык неродной, ключевому слову ANY как квантору дан синтаксический синоним: слово SOME. Например, $= \text{ANY}$ — то же самое, что $= \text{SOME}$:

```
SELECT ename, sal, deptno
FROM    emp
WHERE   ( sal, deptno ) = SOME ( SELECT  MAX ( sal ), deptno
                             FROM    emp
                             GROUP BY deptno )
;
```

6.4.11. Условный оператор EXISTS

Оператор EXISTS применяется к (произвольному) подзапросу с целью определить, возвращает ли в текущем состоянии БД этот подзапрос по меньшей мере одну строку.

Пример. Пусть надо «выдать отделы, в которых есть сотрудники»:

```
SELECT  dname
FROM    dept
WHERE   deptno IN ( SELECT deptno FROM emp )
;
```

То же самое на SQL можно спросить иначе:

```
SELECT  dname
FROM    dept
WHERE   EXISTS ( SELECT * FROM emp WHERE deptno = dept.deptno )
;
```

Это очередной пример избыточности SQL: запросы с IN (SELECT ...) и с EXISTS часто взаимозаменяемы.

Выражение для EXISTS, исходя из своего смысла, *всегда* возвращает только TRUE или FALSE и никогда не приведет к NULL.

В последнем примере подзапрос обладает одной особенностью. Он *связанный* («коррелированный»), то есть содержит обращение к значению, взятому из охватывающего запроса (DEPT.DEPTNO). От этого схема выполнения подзапроса на каждой строке таблицы DEPT будет:

```
(1) SELECT * FROM emp WHERE deptno = 10
(2) SELECT * FROM emp WHERE deptno = 20
...
```

Это может вызвать подозрения в чудовищной неэффективности оператора EXISTS, когда он применен к связанному подзапросу. Хотя тема оптимизации запросов не входит в круг задач настоящего материала, нелишне заметить, что СУБД не обязана *фактически* выполнять все новые подзапросы при выборе очередной строки из DEPT с такой прямолинейностью. Оптимизатор запросов Oracle достаточно разумен, чтобы суметь предложить нередко менее затратную технику обработки. В этом случае он может переформулировать запрос, заменив EXISTS на IN (в то же время в конкретных обстоятельствах подобная замена не всегда приведет к ускорению вычислений).

К сожалению, в других случаях оптимизатор может проявить себя далеко не так разумно, поэтому программист пока еще вынужден контролировать его решения, имея дело с ответственными запросами.

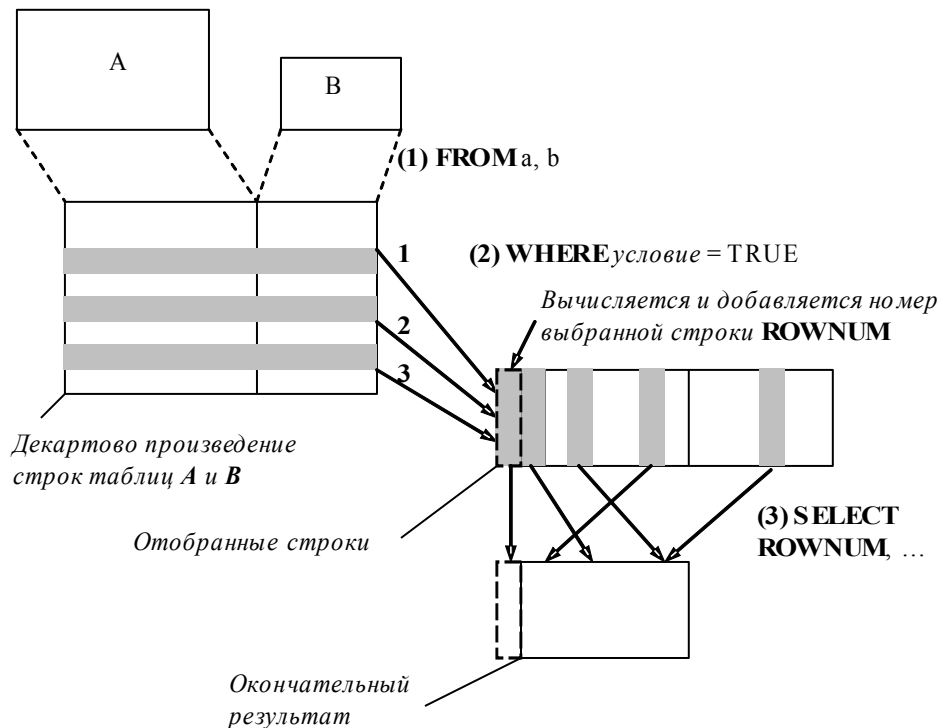
6.4.12. Псевдостолбец ROWNUM и особенности его использования

Выполнение фразы WHERE может сопровождаться порождением значений для «псевдостолбца» ROWNUM. Эта фактически системная функция без параметров выдает номер строки *в результате того SQL-запроса*, где он фигурирует. Пример:

```
SQL> SELECT ROWNUM, ename FROM emp WHERE sal > 1500;
```

ROWNUM	ENAME
1	ALLEN
2	JONES
3	BLAKE
4	CLARK
5	SCOTT
6	KING
7	FORD

В логической последовательности обработки предложения SELECT очередное значение для ROWNUM порождается СУБД при отборе очередной строки фразой WHERE и становится доступным для всех последующих шагов обработки. Вот как может выглядеть логическая схема обработки предложения SELECT ROWNUM, ... FROM a, b WHERE *условие*:



Поэтому на ROWNUM можно сослаться в самой фразе WHERE и всех далее исполняемых, например:

```
SELECT ROWNUM, ename, sal FROM emp WHERE ROWNUM <= 4;
```

Причем несмотря на то, что формально допустимы любые сравнения, смысл приобретают лишь сравнения вида:

```
ROWNUM = 1
ROWNUM < n
ROWNUM <= n
```

Рекурсивный характер исполнения SELECT с CONNECT BY вносит формальную сумятицу в последнее утверждение, позволяя получать в ответе более одной строки при наличии в WHERE условия ROWNUM = n со значениями n = 0 (sic !), 2, 3 и так далее:

```
SELECT ROWNUM, dname FROM dept CONNECT BY ROWNUM = 0;
SELECT ROWNUM, dname FROM dept CONNECT BY ROWNUM = 1;
SELECT ROWNUM, dname FROM dept CONNECT BY ROWNUM = 2;
```

Однако же в таких формально допустимых формулировках мало смысла.

Еще одно следствие состоит в том, что наличие в окончательном ответе значений ROWNUM не предполагает автоматического перечисления их в каком-нибудь порядке. Последний запрос выдаст не более четырех (формально произвольных) строк о сотрудниках с указанием зарплаты. Повторение этого запроса теоретически может вернуть те же строки, но в ином порядке. К тому же, порядок можно сбить искусственно:

```
SQL> SELECT ROWNUM, ename, sal FROM emp WHERE ROWNUM <= 4
      2 ORDER BY sal;
```

ROWNUM	ENAME	SAL
1	SMITH	800
3	WARD	1250
2	ALLEN	1600

Мы *не* получим перечисления четырех наиболее оплачиваемых сотрудников.

Псевдостолбец ROWNUM используется в Oracle не для моделирования данных приложения, а в технических целях при составлении запросов к БД. Например, он находит применение в запросах типа «первая *N*-ка», «выдать первые *N* строк» по заданному критерию. Другой пример — это добавление в условие выражение формулировки AND ROWNUM <= 1, используемое для искусственного сужения объема выдачи до максимума одной строки. О применении этого метода в построении скалярных подзапросов говорилось выше.

6.5. Фраза SELECT и функции в предложении SELECT

Фраза SELECT — вторая, вместе с FROM, обязательная для каждого предложения SELECT. Ее назначение состоит в формировании столбцов таблицы — окончательного результата выполнения запроса. Типично она сохраняет количество строк, поступивших ей на входе от предшествующих фраз, и занимается только переформулированием столбцов, но в некоторых случаях она способна вдобавок и сократить количество строк.

Обычно состав фразы SELECT — список через запятую выражений для столбцов окончательного ответа. В отдельных случаях у такой структуры могут существовать свои особенности.

6.5.1. Сокращенная запись для «всех столбцов таблицы»

Если делается запрос по одной таблице-источнику данных и требуется выдать все поля строк этой таблицы без изменений, вместо списка выражений во фразе SELECT можно указать символ *:

```
SELECT * FROM dept;
```

Символ * может быть предварен именем таблицы:

```
SELECT dept.* FROM dept;
```

В данном случае в этом нужды нет, но если бы источников данных было несколько, такое уточнение было бы оправдано.

Два последних примера равносильны и выдадут то же, что и следующая формулировка:

```
SELECT dept.deptno, dept.dname, dept.loc FROM dept;
```

Еще пример. Следующие два предложения равносильны:

```
SELECT emp.ename, dept.deptno, dept.dname, dept.loc  
FROM dept, emp  
WHERE dept.deptno = emp.deptno  
;
```

```
SELECT emp.ename, dept.*  
FROM dept, emp  
WHERE dept.deptno = emp.deptno  
;
```

Символ * (неуточненный именем) не связан с единичностью источника и означает «все столбцы частичного результата обработки, пришедшие на вход фразе SELECT, без каких-либо преобразований». Пример употребления в запросе к двум таблицам:

```
SELECT *  
FROM dept, emp
```

```
WHERE dept.deptno = emp.deptno
;
```

Использование SELECT * может показаться привлекательным в силу экономности записи, однако стоит напомнить, что это нереляционная конструкция, так как она полагается на порядок столбцов в таблице, в то время как в реляционной модели атрибуты в отношении порядка не имеют и допускают обращение к ним только по названиям. В предположении порядка столбцов (тем более неявном) кроется определенный риск, так как таблицы в Oracle допускают добавление и удаление столбцов, из-за чего запросы в программе могут потерять синтаксическую корректность или, хуже того, изменить смысл.

В технических запросах (не в приложении) конструкцией SELECT * можно пользоваться свободнее.

6.5.2. Выражения во фразе SELECT

Пример, когда во фразе SELECT приводятся выражения, а не имена столбцов:

```
SELECT
    ename
  , ' earns '
  , ( sal + NVL ( comm, 0 ) ) / 1000000
  , ' million dollars per month '
FROM emp
;
```

Первый по порядку столбец будет содержать разные имена сотрудников, второй и четвертый — постоянные значения, а третий — разные результаты оценки числового выражения.

6.5.3. Именование столбцов в результате запроса

Если не предпринять специальных мер, столбцы в таблице-результате именуется автоматически (чаще всего на основе имен столбцов запрошенных таблиц). При желании программист может потребовать СУБД назвать столбец по-своему, указав имя через пробел после формулировки выражения. (Имеется в виду «обобщенный пробел», который может состоять фактически из нескольких знаков пробела, табуляции или переходов на новую строку). Примеры:

```
SELECT ename, sal salary FROM emp;

SELECT ename "Сотрудники", sal "Зарплата" FROM emp;

SELECT
    SUM ( comm ) / COUNT ( * ) "Усреднение по всем сотрудникам"
FROM emp
;
```

Правила выбора и записи имен столбцов те же, что и для таблиц в БД.

Вместо обобщенного пробела можно с равным успехом использовать связку AS:

```
SELECT ename AS "Сотрудники", sal AS "Зарплата" FROM emp;
```

Использовать пробел или ключевое слово AS — дело вкуса и здравого смысла программиста. Ключевое слово AS добавляет тексту запроса переносимости.

6.5.4. Уточнение DISTINCT (UNIQUE)

Пусть нужно узнать, в каких отделах есть сотрудники:

```
SQL> SELECT deptno FROM emp;
```

DEPTNO
20
30
30
20
30
30
10
20
10
30
20
30
20
10

Это неудобно большим количеством повторений даже при такой скромной выборке, как в данном случае. Ключевое слово **DISTINCT** позволяет не выводить в окончательный ответ повторяющиеся строки:

```
SQL> SELECT DISTINCT deptno FROM emp;
```

DEPTNO
30
20
10

Особенности технического отсева повторений в результате употребления слова **DISTINCT**

- Он требует дополнительного времени на свое осуществление.
- Он не нужен, когда строки гарантированно разные (например, отбираются первичные ключи таблицы).
- До версии 10 его осуществление имеет побочный эффект в виде упорядочивания строк результата. Хотя он был и «вне закона», некоторые программисты им пользовались, «потому что так было всегда». С версии 10 побочное упорядочение результата пропало, так что заставить СУБД обрабатывать **DISTINCT** по-старому все еще можно, но уже искусственным путем.

Ограничения использования:

- отсев дубликатов невозможен при наличии столбцов с типами LOB, LONG и некоторых других.

Чтобы подчеркнуть отсутствие отсева повторений, в противовес **DISTINCT** можно явно указать умолчательное **ALL**:

```
SELECT ALL job, sal FROM emp;
```

На равных правах со словом **DISTINCT** во фразе **SELECT** Oracle допускает указание **UNIQUE**. Так, один из предшествующих запросов может быть записан иначе с полным сохранением смысла:

```
SELECT UNIQUE deptno FROM emp;
```

С реляционной точки зрения **DISTINCT** (**UNIQUE**) должно было бы не то что подразумеваться по умолчанию, но «быть» по умолчанию единственно возможным.

6.5.4.1. Учет отсутствующих значений при отсеве дубликатов

Отсутствующие значения в полях строк при *внутреннем*, техническом сравнении с уже отобранными в процессе отсева дубликатов строками считаются *равными друг другу*:

```
SQL> SELECT DISTINCT comm, job FROM emp;
```

COMM	JOB
	CLERK
300	SALESMAN
	PRESIDENT
0	SALESMAN
500	SALESMAN
	MANAGER
1400	SALESMAN
	ANALYST

8 rows selected.

Такое поведение противоречит правилу, согласно которому *явно указанное в запросе* сравнение с отсутствующим значением дает отсутствующий логический результат (NULL, то есть не TRUE и не FALSE), смысл которого — «сравниваемые величины не равны». Это же исключение из общего правила сравнения значений в SQL имеет место при группировке GROUP BY и при операции UNION результатов SELECT (приводятся ниже). Это вынужденная мера: не будь этого исключения, SQL значительно потерял бы в своей практической ценности.

6.5.5. Агрегатные функции в предложении SELECT

Ниже перечисляются некоторые примеры популярных стандартных агрегатных (обобщающих) функций, аргументом для которых выступает столбец значений. Функции COUNT, MIN, MAX, LISTAGG работают на основных встроенных типах: чисел, строк текста, моментов времени, интервалов времени, а также на объектных (MIN и MAX — не всегда, и за исключением LISTAGG). Остальные, за вычетом считанного количесива других, здесь не упомянутых, работают *только* на числовых выражениях и используются для числового анализа данных.

6.5.5.1. Функция COUNT

COUNT используется для подсчета *строк*: вообще в таблице, или с указанными значениями в вычисляемом столбце. Примеры:

```
SELECT COUNT ( * ) FROM emp /* количество строк */;
```

```
SELECT COUNT ( comm ) FROM emp /* количество строк со значениями в столбце */;
```

Подсчет строк в таблице из БД, как выше, — всего лишь частный случай. COUNT (*) не возбраняется применять и в запросе к нескольким источникам данных.

Подсчет строк с указанными значениями в столбце принимает в расчет только строки, где в соответствующем столбце поле имеется значение (в последнем запросе таких окажется четыре).

Указание в выражении-аргументе для агрегатной функции слова DISTINCT (или UNIQUE) позволит подсчитать количество строк с *разными* значениями в столбце:

```
SELECT COUNT ( DISTINCT deptno ) FROM emp /* строки с разными значениями */;
```

Формально это же уточнение DISTINCT допускается и во всех остальных агрегатных функциях, но не всегда при этом оно имеет смысл. Например, в SELECT MAX (DISTINCT ...), SELECT SUM (

DISTINCT ...), и так далее, добавление DISTINCT ничего не дает кроме усложнения текста (а, возможно, и вычисления).

6.5.5.2. Функции MIN и MAX

Выдают минимальное и максимальное значения из наличествующих в столбце. Примеры следуют ниже.

«Выдать максимальный оклад сотрудников»:

```
SELECT MAX ( sal ) FROM emp;
```

«Выдать минимальный оклад сотрудников из Далласа»:

```
SELECT MIN ( sal )  
FROM emp  
WHERE deptno IN ( SELECT deptno FROM dept WHERE LOC = 'DALLAS' )  
;
```

«Сколько сотрудников пришло в первый день?»:

```
SELECT COUNT ( * )  
FROM emp  
WHERE  
    TRUNC ( hiredate )  
    = TRUNC ( ( SELECT MIN ( hiredate ) FROM emp ) )  
;
```

«Какова разница между максимальным и минимальным окладами в центах?»:

```
SELECT ( MAX ( sal ) - MIN ( sal ) ) * 100 FROM emp;
```

6.5.5.3. Функция LISTAGG

С версии 11.2 имеется функция LISTAGG, возвращающая в виде строки VARCHAR2 склеенные требуемым разделителем значения из столбца. Примеры:

«Выдать список имен сотрудников через запятую в порядке убывания зарплат каждого»:

```
SELECT  
    LISTAGG ( ename, ', ' ) WITHIN GROUP ( ORDER BY sal DESC ) namelist  
FROM emp  
;
```

«Выдать список комиссионных сотрудников через запятую в порядке их появления в организации»:

```
SELECT  
    LISTAGG ( comm, ', ' ) WITHIN GROUP ( ORDER BY hiredate ) namelist  
FROM emp  
;
```

Обратите внимание на выходящую из общего ряда формулировку функции LISTAGG, всегда требующей уточнения WITHIN GROUP.

Для получения результата в виде строки текста значения из столбца автоматически приводятся к символьному виду функцией TO_CHAR. Ввиду того, что результат возвращается в виде VARCHAR2, попытка получить список значений, занимающий в символьном выражении более 4000 байт приведет к ошибке исполнения (но даже в версии 12).

6.5.5.4. Другие примеры

Пример использования функции суммирования значений SUM.

«Выдать сумму разных значений окладов сотрудников из Далласа»:

```
SELECT SUM ( DISTINCT sal )
FROM   emp
WHERE  deptno IN ( SELECT deptno FROM dept WHERE LOC = 'DALLAS' )
;
```

Пример подсчета среднего значения из наличествующих в столбце.

«Выдать должности, для которых оклад выше среднего»:

```
SELECT DISTINCT job
FROM   emp
WHERE  sal > ( SELECT AVG ( sal ) FROM emp )
;
```

6.5.5.5. Общие правила для стандартных агрегатных функций

Для стандартных агрегатных функций выполняются общие правила вычисления.

- Если для каких-то строк столбца оценка выражения приводит NULL, агрегатная функция эти строки игнорирует, она обобщает данные только *существующих* значений (не-NULL).
- За исключением COUNT, если *все* значения в столбце отсутствуют (NULL) или же если столбец пуст, то будет отсутствовать (NULL) результат обобщения.
- Исключение: COUNT всегда возвращает значение, в крайнем случае 0 (столбец отсутствующих значений или же пустое множество строк).

Исходя из этого, следующие выражения при обращении к EMP в общем случае *не* равнозначны:

```
AVG ( NVL ( comm, 0 ) )
NVL ( AVG ( comm ), 0 )
SUM ( comm ) / COUNT ( * )
```

Употреблять агрегатные функции в запросе следует с осторожностью, отдавая себе отчет об их поведении на пустом множестве значений или же строк. Исключение, сделанное для COUNT, связано с особым смыслом этой функции: COUNT всегда подсчитывает именно строки. Если об этом забыть, поведение COUNT становится неинтуитивным. В самом деле, легко предположить, что COUNT — не самостоятельная по сути операция, сводимая к SUM. Например, COUNT (*) по сути равносильно SUM (1), а COUNT (*выражение*) по сути равносильно SUM (CASE WHEN *выражение* IS NOT NULL THEN 1 END), однако на пустом множестве SQL (и в стандарте, и в исполнении Oracle) эти формулировки оценивает по-разному. Когда же полагать, как это и заявлено в определении, что COUNT считает строки, то количество строк не может отсутствовать, но может быть равным нулю.

Есть также формально-синтаксические запреты на употребление. Если в предложении SELECT нет GROUP BY и если для формирования столбцов результата применяются агрегатные функции, то использование в столбцах результата имен столбцов таблиц-источников *вне* агрегатных функций запрещено.

Примеры. Следующее предложение *некорректно* (ошибочно) синтаксически:

```
SELECT COUNT ( * ), ename FROM emp;
```

Следующее предложение корректно:

```
SELECT SUM ( comm ) / COUNT ( * ) + 123 FROM emp;
```

Упражнение. Ответить прямой речью, что выдаст последний запрос. Сравнить выражения AVG (comm) и SUM (comm) / COUNT (comm).

6.5.6. Аналитические функции

Аналитические функции — это *нескалярные* функции (за исключением аналитических статистических, скалярных по результату), которые в отличие от стандартных агрегатных могут употребляться *только* во фразах SELECT и ORDER BY, так как применяются к уже отобранному результату (см. выше схему выполнения предложения SELECT). Свое название получили по той причине, что позволяют средствами SQL (в Oracle) строить запросы, анализирующие данные в БД. Являются вариацией «оконных функций», вошедших в SQL:2003; другая вариация реализована фирмой IBM в DB2.

Функции этой категории иногда называют «функциями OLAP» ввиду того, что они хорошо подходят для систем типа OLAP (on-line analytical processing), аналитических систем и «аналитических баз данных».

В Oracle они могут быть следующих видов:

- (a) функции ранжирования;
- (b) статистические функции для плавающего интервала;
- (c) функции подсчета долей;
- (d) статистические функции LAG/LEAD с запаздывающим/опережающим аргументом;
- (e) статистические функции (линейная регрессия и т. д.).

Далее по очереди приводятся примеры употребления аналитических функций каждого из этих видов.

«Раздать сотрудникам места по мере убывания или возрастания их зарплат»:

```
SELECT
  ename
, sal
, ROW_NUMBER ( ) OVER ( ORDER BY sal DESC ) AS row_number_desc
, ROW_NUMBER ( ) OVER ( ORDER BY sal )      AS row_number_asc
, RANK ( )      OVER ( ORDER BY sal )        AS rank
, DENSE_RANK ( ) OVER ( ORDER BY sal )       AS dense_rank
FROM emp
;
```

Ответ:

ENAME	SAL	ROW_NUMBER_DESC	ROW_NUMBER_ASC	RANK	DENSE_RANK
SMITH	800	14	1	1	1
JAMES	950	13	2	2	2
ADAMS	1100	12	3	3	3
MARTIN	1250	11	4	4	4
WARD	1250	10	5	4	4
MILLER	1300	9	6	6	5
TURNER	1500	8	7	7	6
ALLEN	1600	7	8	8	7
CLARK	2450	6	9	9	8
BLAKE	2850	5	10	10	9
JONES	2975	4	11	11	10
SCOTT	3000	3	12	12	11
FORD	3000	2	13	12	11
KING	5000	1	14	14	12

Как видно, разница в поведении проявляется на данных, где критерий определения места оказывается одинаковым у нескольких сотрудников. ROW_NUMBER в таких случаях места раздает случайно. Например в версии 9 СУБД на этот запрос выдавала Скотту и Форду второе и третье места, а Мартину и

Варду — десятое и одиннадцатое. Функции же RANK и DENSE_RANK на одинаковом показателе критерия присваивают строкам одно и то же место с той разницей, что в случае DENSE_RANK следующее по величине критерия место выдается по порядку, а в случае RANK — с пропуском за счет возникших повторений.

Обратите внимание на конструкцию ORDER BY в определении функций ранжирования. Она принадлежит формулировке функций, уточняет способ их вычисления, и никоим образом не задает порядка строк конечного результата (это видно из приведенного ответа). Установление порядка строк конечного результата осуществляется только фразой ORDER BY предложения SELECT. (Конкретно слова ORDER BY здесь, вероятно, выбраны разработчиками Oracle SQL из соображения общности. Они выглядят более убедительно в нижеследующих примерах аналитических функций; в функциях же ранжирования они, по сути, используются просто ради возможности указать выражение для критерия упорядочения, а не для наведения порядка как такового. При ином развитии событий вместо ORDER BY здесь могли бы быть взяты какие-нибудь другие слова.)

«Растущий итог» выплат на зарплату по мере приема сотрудников на работу»:

```
SELECT
  ename
, sal
, hiredate
, SUM ( sal ) OVER
  ( ORDER BY hiredate
    RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW )
  AS sum_over_range
FROM emp
;
```

Ответ:

ENAME	SAL	HIREDATE	SUM_OVER_RANGE
SMITH	800	17.12.80	800
ALLEN	1600	20.02.81	2400
WARD	1250	22.02.81	3650
JONES	2975	02.04.81	6625
BLAKE	2850	01.05.81	9475
CLARK	2450	09.06.81	11925
TURNER	1500	08.09.81	13425
MARTIN	1250	28.09.81	14675
KING	5000	17.11.81	19675
JAMES	950	03.12.81	23625
FORD	3000	03.12.81	23625
MILLER	1300	23.01.82	24925
SCOTT	3000	19.04.87	27925
ADAMS	1100	23.05.87	29025

Заметьте, что Джеймс и Форд поступили на работу одновременно, и поэтому значение общей суммы зарплат у них одинаковое. В то же время смысл такого суммирования не совсем ясен. Более понятен запрос, где вместо слова RANGE указать ROWS. Если это сделать, Джеймс и Форд «получат» разные суммы, но снова в произвольном порядке (значения HIREDATE как критерия упорядочения у них одинаковые).

Помимо агрегатных функций со «стандартным» оформлением допустимо употребить и своеобразно записываемую функцию NTH_VALUE (существует с версии 11.2). Она позволяет обратиться к *n*-му сверху или же снизу значению в [упорядоченном] столбце, определяемом интервалом (окошком, window). Ниже сформулирован запрос на основе предыдущего, но на интервалах с фиксированной верхней границей (первый сотрудник в организации) и плавающей нижней (очередной сотрудник по мере поступления на работу) подсчитывается не общая сумма зарплат, как раньше, а выдается всякий раз предыдущая зарплата:

```
SELECT
  ename
```

```

, sal
, NTH_VALUE ( sal, 2 ) FROM LAST OVER
(
ORDER BY hiredate
RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
) AS previous_over_range
FROM emp
;

```

Ответ:

ENAME	SAL	PREVIOUS_OVER_RANGE
SMITH	800	
ALLEN	1600	800
WARD	1250	1600
JONES	2975	1250
BLAKE	2850	2975
CLARK	2450	2850
TURNER	1500	2450
MARTIN	1250	1500
KING	5000	1250
JAMES	950	950
FORD	3000	950
MILLER	1300	3000
SCOTT	3000	1300
ADAMS	1100	3000

Сделанное в запросе, предшествующем последнему, замечание по поводу замены RANGE BETWEEN на ROWS BETWEEN остается в силе.

Еще одно замечание касается упрощения формулировки OVER. Она разрешает отказаться от указания не только интервала/окошка, но, в некоторых случаях, и от задания порядка. Вот как можно получить отношение оклада каждого сотрудника к максимальному:

```

SELECT
  ename
, sal
, sal / MAX ( sal ) OVER ( )
  AS salary_ratio
FROM emp
;

```

Тот же ответ получим, разделив SAL на результат скалярного подзапроса (SELECT MAX (sal) FROM emp), однако в таком случае Oracle предложит менее эффективный план исполнения (выполнит лишний просмотр EMP).

Упражнение. Выдать оклады сотрудников и отношение каждого оклада к среднему по организации.

«Доли зарплаты сотрудников в общей сумме зарплат»:

```

SELECT
  ename
, sal
, RATIO_TO_REPORT ( sal ) OVER ( ) AS ratio_to_report
FROM emp
;

```

Ответ:

ENAME	SAL	RATIO_TO_REPORT
SMITH	800	.027562446
ALLEN	1600	.055124892
WARD	1250	.043066322
JONES	2975	.102497847
MARTIN	1250	.043066322

BLAKE	2850	.098191214
CLARK	2450	.084409991
SCOTT	3000	.103359173
KING	5000	.172265289
TURNER	1500	.051679587
ADAMS	1100	.037898363
JAMES	950	.032730405
FORD	3000	.103359173
MILLER	1300	.044788975

«Изменение зарплаты сотрудника по отношению к предшественнику по мере приема на работу»:

```
SELECT
  ename
, sal
, sal - LAG ( sal, 1 ) OVER ( ORDER BY hiredate ) delta
FROM emp
;
```

Ответ:

ENAME	SAL	DELTA
SMITH	800	
ALLEN	1600	800
WARD	1250	-350
JONES	2975	1725
BLAKE	2850	-125
CLARK	2450	-400
TURNER	1500	-950
MARTIN	1250	-250
KING	5000	3750
JAMES	950	-4050
FORD	3000	2050
MILLER	1300	-1700
SCOTT	3000	1700
ADAMS	1100	-1900

Обратите внимание на разумное поведение аналитических функций (вообще) на границах упорядоченных множеств данных (строка со Смитом). Неприятность в другом: NULL, который порождает Oracle в своем ответе, имеет здесь смысл «значение не применимо», а не «неизвестно», как хотелось бы (обсуждение разницы приводилось выше).

```
SELECT
  REGR_SLOPE ( sal, comm ) AS slope
, REGR_AVGX ( sal, comm ) AS avgsal
, REGR_AVGY ( sal, comm ) AS avgcomm
FROM emp
;
```

Ответ:

SLOPE	AVGSAL	AVGCOMM
-.20642202	550	1400

Названия функций для имеющихся прочих видов регрессии приведены в документации по Oracle. Обратите внимание на вероятную формальность этого запроса, если не предположить, что связь между зарплатой и комиссионными в жизни вдруг существует, в результате чего запрос приобретает смысл. Функции регрессии — единственные, требующие в качестве аргументов два столбца.

6.5.7. Выражение типа ссылка на курсор

Во фразе SELECT (а также в качестве аргумента функции — в составе любого выражения) можно использовать выражение типа «ссылка на курсор». Оно строится с помощью функции CURSOR, аргументом которой передается другое предложение SELECT, и возвращает скаляр — ссылку на курсор. Хотя в SQL*Plus эту функцию и разрешено задействовать, основное применение ей — в программной обработке результатов предложения SELECT. Пример в SQL*Plus:

```
SELECT
  dname
, CURSOR ( SELECT ename FROM emp WHERE emp.deptno = dept.deptno )
FROM dept
;
```

В качестве *элемента* более общего выражения SQL курсорное выражение может войти только будучи предъявленным аргументом какой-нибудь функции. Ниже приводится пример передачи данных конструктору типа XMLTYPE через ссылку на курсор. Результат, объект типа XMLTYPE, можно использовать для построения более сложного выражения, однако в примере этого не сделано:

```
SELECT
  XMLTYPE ( CURSOR ( SELECT * FROM emp ) ) AS "Employees data by XML:"
FROM
  dual
;
```

В SQL фирмы Oracle *нет* ссылки на курсор отсутствует. Он имеется только в PL/SQL, где, в частности, может быть использован для описания аргументов при написании функций, обращение к которым будет выполняться, в том числе, в выражениях SQL.

6.6. Соединение фраз SELECT и FROM фразами PIVOT/UNPIVOT

Версия Oracle 11 позволила дополнить фразу FROM подчиненными фразами PIVOT и UNPIVOT, приводящими к автоматическому появлению во фразе SELECT столбцов, «импортированных» из этих конструкций. Обе формулировки предназначены для переформатирования данных таблиц средствами SQL, без программирования. Они удобны для построения отчетов и анализа имеющихся данных.

6.6.1. Разворачивание данных в столбцы указанием PIVOT

Сочетание SELECT ... FROM ... PIVOT ... позволяет развернуть данные одного столбца в отдельные столбцы конечного результата.

Рассмотрим для начала запрос о наличии в разных отделах сотрудников на разных должностях:

```
SQL> SELECT job, deptno FROM emp;
```

JOB	DEPTNO
CLERK	20
SALESMAN	30
SALESMAN	30
MANAGER	20
SALESMAN	30
MANAGER	30
MANAGER	10
ANALYST	20
PRESIDENT	10
SALESMAN	30
CLERK	20
CLERK	30
ANALYST	20
CLERK	10

В каждом отделе имеется ноль или более сотрудников на каждой из вообще существующих должностей. Данные об их количестве удобно представить, посвятив каждому департаменту отдельный столбец:

```
SQL> SELECT *
  2> FROM   ( SELECT job, deptno FROM emp )
  3> PIVOT  ( COUNT ( * ) FOR deptno IN ( 10, 20, 30, 40 ) );
```

JOB	10	20	30	40
CLERK	1	2	1	0
SALESMAN	0	0	4	0
PRESIDENT	1	0	0	0
MANAGER	1	1	1	0
ANALYST	0	2	0	0

Внутренняя отработка такого предложения осуществляется как при группировке GROUP BY job (см. ниже), но приводит к появлению дополнительных столбцов вместо, казалось бы, указанного DEPTNO. Вот как мог бы выглядеть аналог последнего запроса в версиях Oracle до 11:

```
SELECT
  job
, COUNT ( CASE WHEN deptno = 10 THEN 1 END ) "10"
, COUNT ( CASE WHEN deptno = 20 THEN 1 END ) "20"
, COUNT ( CASE WHEN deptno = 30 THEN 1 END ) "30"
, COUNT ( CASE WHEN deptno = 40 THEN 1 END ) "40"
FROM   ( SELECT job, deptno FROM emp )
GROUP BY job
;
```

Именно так Oracle и обработает запрос с PIVOT (по крайней мере в версии 11), но форма с PIVOT приводит к краткости и определенной выразительной гибкости.

Пониманию используемой в Oracle техники разворачивания способствует рассмотрение следующих примеров. Примеры показывают, что выбор внутренней группировки при отработке PIVOT определяется на основе: (а) структуры исходного источника данных, (б) вида использованного агрегирования и (в) конструкции FOR:

```
SELECT * FROM ( SELECT job, deptno FROM emp ) -- группировка по значениям JOB
PIVOT ( COUNT ( * ) AS employees FOR ( deptno ) IN ( 10, 20, 30, 40 ) )
;
SELECT * FROM ( SELECT job, deptno FROM emp ) -- обобщение по всем строкам
PIVOT ( COUNT ( job ) AS employees FOR ( deptno ) IN ( 10, 20, 30, 40 ) )
;
SELECT * FROM ( SELECT job, deptno FROM emp ) -- группировка по значениям JOB
PIVOT ( COUNT ( deptno ) AS employees FOR ( deptno ) IN ( 10, 20, 30, 40 ) )
;
SELECT * FROM ( SELECT job, deptno FROM emp ) -- обобщение по всем строкам
PIVOT ( COUNT ( * ) AS employees FOR ( deptno, job )
      IN ( ( 10, 'CLERK' ), ( 30, 'ANALYST' ) ) )
;
SELECT * FROM ( SELECT job, deptno, comm FROM emp ) -- группировка по <JOB, COMM>
PIVOT ( COUNT ( * ) AS employees FOR ( deptno ) IN ( 10, 20, 30, 40 ) )
;
SELECT * FROM ( SELECT job, deptno, comm FROM emp ) -- группировка по <COMM>
PIVOT ( COUNT ( job ) AS employees FOR ( deptno ) IN ( 10, 20, 30, 40 ) )
;
SELECT * FROM ( SELECT job, deptno, comm FROM emp ) -- группировка по <JOB, COMM>
PIVOT ( COUNT ( deptno ) AS employees FOR ( deptno ) IN ( 10, 20, 30, 40 ) )
;
```

Серым фоном помечены столбцы, выбранные для группировки при вычислении ответа.

Несколько примеров составления запросов и способов оформления. Выдать только данные по отделам 10 и 30:


```

SELECT *
FROM   ( SELECT job, deptno FROM emp )
PIVOT  ( COUNT ( * ) FOR deptno IN ( 10, 30 ) )
;

```

То же самое:

```

SELECT job, "10", "30"
FROM   ( SELECT job, deptno FROM emp )
PIVOT  ( COUNT ( * ) FOR deptno IN ( 10, 20, 30, 40 ) )
;

```

Количества разных должностей без уточнения по отделам:

```

SELECT *
FROM   ( SELECT job, deptno FROM emp )
PIVOT  ( COUNT ( job ) FOR deptno IN ( 10, 30 ) )
;

```

Сообщение столбцам в ответе необходимых имен делается, как во фразе SELECT:

```

SELECT *
FROM   ( SELECT job, deptno FROM emp )
PIVOT  ( COUNT ( * ) FOR deptno IN ( 10 tenth, 30 thirtieth ) )
;

```

Выдача сумм окладов сотрудников по каждой должности в указанных отделах:

```

SELECT *
FROM   ( SELECT job, deptno, sal FROM emp )
PIVOT  ( SUM ( sal ) FOR deptno IN ( 10, 20, 30, 40 ) )
;

```

Выдача сразу двух агрегатов (суммы зарплат и количества сотрудников):

```

SELECT *
FROM   ( SELECT job, deptno, sal FROM emp )
PIVOT  ( SUM ( sal ), COUNT ( * ) c FOR deptno IN ( 10, 20, 30, 40 ) )
;

```

Сформировать столбцы для сотрудников 10-го отдела с окладами 5000 и 1300 и сотрудников 30-го отдела с окладами 1300:

```

SELECT *
FROM   ( SELECT job, deptno, sal FROM emp )
PIVOT  (
    COUNT ( * ) c
    FOR ( deptno, sal ) IN ( ( 10, 5000 ), ( 10, 1300 ), ( 30, 1250 ) )
);

```

Во фразе PIVOT также предусмотрены конструкции для разворачивания данных не в столбцы, а в документы XML. Там количество «столбцов» в разворачивании (элементов XML) в противовес таблице определяется по факту.

Пример из каждодневной практики. Среди объектов БД, доступных обращению пользователя, узнать количества для типов TABLE, VIEW, SYNONYM, INDEX, TYPE и JAVA CLASS:

```

SELECT *
FROM   ( SELECT owner, object_type FROM all_objects )
PIVOT  ( COUNT ( * ) FOR object_type IN
        ( 'TABLE', 'SYNONYM', 'VIEW', 'INDEX', 'JAVA CLASS', 'TYPE' ) )
;

```

6.6.2. Сворачивание данных в столбец указанием UNPIVOT

Фраза UNPIVOT выполняет действие, содержательно противоположное фразе PIVOT.

Создадим таблицу по запросу выше:

```
CREATE TABLE total AS
SELECT *
FROM   ( SELECT job, deptno FROM emp )
PIVOT  ( COUNT ( * ) FOR deptno IN ( 10, 20, 30, 40 ) )
;
```

Выдача данных со сворачиванием показателей в один столбец:

```
SQL> SELECT *
      2 FROM   total
      3 UNPIVOT ( jobcount FOR deptno IN ( "10", "20", "30", "40" ) )
      4 ;
```

JOB	DE	JOBCOUNT
-----	--	-----
CLERK	10	1
CLERK	20	2
CLERK	30	1
CLERK	40	0
SALESMAN	10	0
SALESMAN	20	0
SALESMAN	30	4
SALESMAN	40	0
PRESIDENT	10	1
PRESIDENT	20	0
PRESIDENT	30	0
PRESIDENT	40	0
MANAGER	10	1
MANAGER	20	1
MANAGER	30	1
MANAGER	40	0
ANALYST	10	0
ANALYST	20	2
ANALYST	30	0
ANALYST	40	0

От выдачи JOBCOUNT можно отказаться, указав вместо SELECT * ... формулировку SELECT job, deptno

Заметьте, что агрегат COUNT в отличие от всех остальных всегда возвращает значение, хотя бы 0. Если бы столбец JOBCOUNT подсчитывался иначе, и в нем бы оказались пропуски (NULL), закономерен был бы вопрос, как эти пропуски учитывать при сворачивании данных. Две умышленно возможные схемы поведения обозначаются уточнениями UNPIVOT INCLUDE NULLS и UNPIVOT EXCLUDE NULLS.

Подобное сворачивание в столбец (иначе переворачивание, «транспонирование») таблицы TOTAL позволяет ответить на вопросы типа: «В каких отделах число продавцов больше 10 ?», или даже «больше 10%». Иначе SQL этого делать не умеет.

6.7. Фраза ORDER BY предложения SELECT

Фраза ORDER BY дает единственно законный способ получить упорядоченный результат запроса на SQL, за исключением запросов с CONNECT BY.

6.7.1. Простейшая сортировка

```
SELECT ename, sal FROM emp ORDER BY sal;
```

Строки ответа сортируются по возрастанию величины зарплаты. Убывающий порядок должен задаваться явочным путем с помощью слова DESC:

```
SELECT ename, hiredate FROM emp ORDER BY hiredate DESC;
```

В противовес этому, ради ясности можно сослаться на возрастающий порядок с помощью формально необязательного слова ASC.

Упорядочение строк ответа по нескольким столбцам задается перечислением в ORDER BY столбцов через запятую.

Пусть имеется предложение:

```
SELECT ename, sal FROM emp;
```

Допустимые варианты формулировок фразы ORDER BY:

```
ORDER BY ename DESC
ORDER BY sal ASC, ename DESC
ORDER BY sal, ename
ORDER BY emp.sal
ORDER BY emp.ename DESC, sal, hiredate DESC
```

6.7.2. Упорядочение по значению выражения

```
SELECT ename FROM emp ORDER BY NVL ( sal, 0 ) + NVL ( comm, 0 );
```

```
SELECT ename, hiredate FROM emp ORDER BY TRUNC ( hiredate ) DESC;
```

Если проставить в качестве значения упорядочения функцию выдачи случайных чисел, фразой ORDER BY можно добиться (псевдо)случайного порядка строк в результате:

```
SELECT ename FROM emp ORDER BY DBMS_RANDOM.VALUE;
```

6.7.3. Указание номера столбца

Если в выражении для упорядочения указано число (формата NUMBER), то оно воспринимается не как признак упорядочения, а как номер столбца во фразе SELECT, по значениями которого следует упорядочить результат:

```
SELECT job, AVG ( sal ) FROM emp
GROUP BY job
ORDER BY 2
;
```

Упражнение. Число формата NUMBER не обязано быть целым, но тогда автоматически будет обрезано функцией TRUNC до целого. Проверить работу запроса и объяснить результат, где вместо 2 указано: (а) '2', (б) 2.1, (в) 2.7, (г) 2f, (д) 2.1f.

Указание числового выражения вместо явного числа приводит к не всегда прогнозируемым результатам. Проверить работу запроса, где указано ORDER BY (SELECT 2 FROM dual); (SELECT 2.1 FROM dual); (SELECT 2f FROM dual); (SELECT 2.1f FROM dual).

Указание номера во фразе ORDER BY может сделать формулировку запроса более надежной, если результат следует упорядочить по столбцу, построенному на основе «полноценного» выражения. Менее надежная, но идентичная по результату формулировка запроса выше:

```
SELECT job, AVG ( sal ) FROM emp
GROUP BY job
ORDER BY AVG ( sal )
;
```

Ее недостаток: повторяя (или исправляя) выражение, программист может ошибиться и в том, что для сложных выражений СУБД может не распознать их идентичность и вычислять дважды (в простых случаях оптимизатор запросов в Oracle двоекратного вычисления делать не будет).

Третий, равносильный по результату вариант формулировки запроса, сохраняет преимущество первой формулировки перед второй, но лишен ее недостатка:

```
SELECT job, AVG ( sal ) avgсал FROM emp
GROUP BY job
ORDER BY avgсал
;
```

Она позволяет СУБД единожды, а не дважды вычислить выражение, устраняет риск ошибки программиста при повторении записи выражения и не опирается на номер столбца. Ее-то и можно рекомендовать для использования в приложении. Интересно, что в ней SQL дает очередной пример собственной непоследовательности: нарушает свою же логическую схему обработки запроса, позволив сослаться в предпоследней по порядку выполнения фразе ORDER BY на название AVGSAL, определенное позже, в завершающей фразе SELECT.

6.7.4. Упорядочение текстовых значений: двоичное и по правилам языка

В отличие от данных других видов, в основе упорядочения строк текста имеется два содержательно разных способа их сравнения: по двоичным кодам отдельных символов и по правилам языка. Требуемый способ определяется параметром СУБД NLS_SORT, допускающим установку индивидуально для сеансов. Проверка:

```
INSERT INTO emp ( empno, ename ) VALUES ( 1111, 'adams' );
ALTER SESSION SET NLS_SORT = BINARY;
SELECT ename FROM emp ORDER BY ename;
```

Результат:

```
ENAME
-----
ADAMS
ALLEN
BLAKE
...
WARD
adams
```

Результат приведен для русских кодировок для Unix/Windows, построенных на основе ASCII. В случае кодировки на основе EBCDIC порядок результата будет фиксированный, но иной. Сравнение текстов на основе кодов символов — наиболее быстрое.

Далее:

```
ALTER SESSION SET NLS_SORT = RUSSIAN;
```

```
SELECT ename FROM emp ORDER BY ename;
```

Результат:

```
ENAME
-----
ADAMS
adams
ALLEN
BLAKE
...
WARD
```

Этот результат соответствует традиционному для русского языка порядку строк, знакомому по словарям докомпьютерных времен. Он достигается определенными дополнительными затратами на обработку. На латинских буквах, как в этом примере, с равным успехом можно было применить NLS_LANG = LATIN.

Далее:

```
ALTER SESSION SET NLS_SORT = RUSSIAN_CI;
```

```
SELECT ename FROM emp ORDER BY ename;
```

Результат:

```
ENAME
-----
adams
ADAMS
ALLEN
BLAKE
...
WARD
```

Указание CI в значении для NLS_LANG расшифровывается как case-insensitive, то есть игнорирование регистра. Отсюда иное расположение «маленького Адамса» в результате, которое, впрочем, не гарантировано (регистр *не* принимается во внимание !) Для чисто латинских букв соответствующее значение записывается как LATIN_CI.

Восстановим данные:

```
ROLLBACK;
```

Специальная функция NLSSORT позволяет указать нужный способ сортировки независимо от установок сеанса:

```
SELECT ename FROM emp ORDER BY NLSSORT ( ename, 'NLS_SORT=RUSSIAN' );
```

Эту функцию можно использовать в любых выражениях, где играет роль порядок присутствующих значений, например:

```
SELECT ename
FROM   emp
WHERE  NLSSORT ( ename, 'NLS_SORT=RUSSIAN' )
BETWEEN NLSSORT ( 'allen', 'NLS_SORT=RUSSIAN' )
AND NLSSORT ( 'KING', 'NLS_SORT=RUSSIAN' )
;
```

Пример с русскими буквами имеет дополнительную окраску в силу особого расположения в кодировочной таблице буквы ё:

```
SQL> ALTER SESSION SET NLS_SORT = BINARY;

SQL> SELECT DECODE ( ROWNUM, 1, 'e', 2, 'ë', 3, 'Ж', 4, 'Ё' ) ltr
      2 FROM dept ORDER BY ltr;
```

```
L
-
Ё
ë
Ж
e
```

```
SQL> ALTER SESSION SET NLS_SORT = RUSSIAN;
```

Session altered.

```
SQL> SELECT DECODE ( ROWNUM, 1, 'e', 2, 'ë', 3, 'Ж', 4, 'Ё' ) ltr
      2 FROM dept ORDER BY ltr;
```

```
L
-
e
Ё
ë
Ж
```

```
SQL> SELECT DECODE ( ROWNUM, 1, 'e', 2, 'ë', 3, 'Ж', 4, 'Ё' ) ltr
      2 FROM dept ORDER BY NLSSORT ( ltr, 'NLS_SORT=BINARY' );
```

```
L
-
Ё
ë
Ж
e
```

(Примеры с русскими буквами отработают правильно, если перед вызовом клиентской программы, а в данном случае это SQL*Plus, установить корректное значение переменной среды окружения OC NLS_LANG).

В условных выражениях (фраза WHERE в SQL или в блоках PL/SQL) принимается во внимание не только значение параметра NLS_SORT, но и NLS_COMP. Последнее может быть BINARY или LINGUISTIC. Если NLS_COMP = LINGUISTIC, решение принимается исходя из значения NLS_SORT. Пример разясняющей последовательности действий:

```
ALTER SESSION SET NLS_COMP = BINARY;
ALTER SESSION SET NLS_SORT = BINARY;
SELECT 'e < ë' FROM dual WHERE 'e' < 'ë';
ALTER SESSION SET NLS_COMP = BINARY;
ALTER SESSION SET NLS_SORT = RUSSIAN;
SELECT 'e < ë' FROM dual WHERE 'e' < 'ë';
ALTER SESSION SET NLS_COMP = LINGUISTIC;
ALTER SESSION SET NLS_SORT = BINARY;
SELECT 'e < ë' FROM dual WHERE 'e' < 'ë';
ALTER SESSION SET NLS_COMP = LINGUISTIC;
ALTER SESSION SET NLS_SORT = RUSSIAN;
SELECT 'e < ë' FROM dual WHERE 'e' < 'ë';
```

Упражнение. Выполнить приведенные выше операции и наблюдать эффект различных комбинаций значений на сравнение величин.

Другие особенности параметров СУБД (сеанса), определяющих различные модели сравнения текстов, приведены в документации по Oracle.

Текущие значения параметров сравнения и сортировки своего сеанса можно посмотреть в таблице словаря-справочника NLS_SESSION_PARAMETERS, выдав:

```
SELECT *
FROM
  nls_session_parameters
WHERE
  parameter IN ( 'NLS_COMP', 'NLS_SORT' )
;
```

6.7.5. Особенности обработки отсутствующих значений при сортировке

Если не указать особо, отсутствующие значения интерпретируются в Oracle как *наибольшие* для соответствующего типа. При сортировке по возрастанию они размещаются в конце, а по убыванию — в начале списка. (В стандарте SQL это не зафиксировано). Влиять на расположение строк с отсутствующими значениями полей можно во фразе ORDER BY с помощью указаний NULLS FIRST и NULLS LAST.

Пример:

```
SELECT  ename, comm
FROM    emp
ORDER BY comm NULLS FIRST
        , sal
;
```

6.8. Отбор порции строк фразой FETCH ROWS

Фраза указывается и исполняется в конце текста и отработки предложения SELECT. Появилась в версии 12, и предназначена для отбора в конечный результат порции строк из начала или середины общего ответа.

Имеет формулировку:

```
[ OFFSET целое { ROW|ROWS } ]
  FETCH { FIRST|NEXT } { целое|целое PERCENT } { ROW|ROWS }
                                     { ONLY|WITH TIES }
```

Слова ROW и ROWS, а также FIRST и NEXT здесь являются синонимами и выбираются по вкусу программиста.

Выдать пять сотрудников с наибольшими зарплатами:

```
SELECT  ename, sal
FROM    emp
ORDER BY sal DESC
FETCH FIRST 5 ROWS ONLY
;
```

Выдать пять процентов сотрудников с наибольшими зарплатами:

```
SELECT  ename, sal
FROM    emp
ORDER BY sal DESC
FETCH FIRST 5 PERCENT ROWS ONLY
;
```

Выдать пять процентов сотрудников с наибольшими зарплатами плюс сколько есть с последней в этом списке по размеру зарплатой:

```

SELECT  ename, sal
FROM    emp
ORDER BY sal DESC
FETCH FIRST 5 PERCENT ROWS WITH TIES
;

```

Выдать следующие пять сотрудников с наибольшими зарплатами после первого:

```

SELECT  ename, sal
FROM    emp
ORDER BY sal DESC
OFFSET 1 ROWS FETCH FIRST 5 ROWS ONLY
;

```

Наличие фразы ORDER BY в таких запросах необязательно, но делает ответ надежнее (в смысловом отношении).

6.9. Фразы GROUP BY и HAVING предложения SELECT

Назначение фразы GROUP BY — сгруппировать строки по указанному общему признаку и выдать сведения, общие для каждой группы. Подчиненная ей фраза HAVING употребляется для отсева по мере необходимости некоторых *групп*, подобно тому как фраза WHERE отсеивает *строки*, полученные из источников данных запроса, по сформулированному условному выражению.

6.9.1. Пример отработки фразы GROUP BY ... HAVING

Рассмотрим запрос:

```

SELECT  deptno
FROM    emp
WHERE    sal > 1000
GROUP BY deptno
HAVING COUNT ( * ) <= 4
ORDER BY deptno DESC
;

```

«Выбрать в убывающем порядке номера отделов, в которых число сотрудников с окладом > 1000 менее пяти»

Запрос намеренно упрощен тем, что не учитывает наличия данных о связи сотрудников и отделов вне таблицы EMP. В действительности, в схеме SCOTT это не так: высказывание «имеются отделы, где нет сотрудников» требует для определения истинности обращения сразу к двум таблицам: EMP и DEPT. Точнее приведенной формулировке на SQL соответствует следующая на естественном языке:

«Выбрать в убывающем порядке номера отделов, в которых есть сотрудники, но оклад > 1000 имеют менее пяти из них»

Ниже отработка фраз GROUP BY и HAVING поясняется серией шагов в соответствии с общей логикой выполнения, упоминавшейся ранее.

6.9.1.1. Промежуточный результат после FROM ... WHERE ...

Из-за наличия в запросе всего одного источника данных фраза FROM всего только выбирает строки из таблицы EMP, настоящую работу выполняет фраза WHERE. Вот ее результат:

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
-------	-------	-----	-----	----------	-----	------	--------

7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	300	30
7521	WARD	SALESMAN	7698	22-FEB-81	1250	500	30
7566	JONES	MANAGER	7839	02-APR-81	2975		20
7654	MARTIN	SALESMAN	7698	28-SEP-81	1250	1400	30
7698	BLAKE	MANAGER	7839	01-MAY-81	2850		30
7782	CLARK	MANAGER	7839	09-JUN-81	2450		10
7788	SCOTT	ANALYST	7566	19-APR-87	3000		20
7839	KING	PRESIDENT		17-NOV-81	5000		10
7844	TURNER	SALESMAN	7698	08-SEP-81	1500	0	30
7876	ADAMS	CLERK	7788	23-MAY-87	1100		20
7902	FORD	ANALYST	7566	03-DEC-81	3000		20
7934	MILLER	CLERK	7782	23-JAN-82	1300		10

6.9.1.2. Промежуточный результат после GROUP BY

Группировка по признаку «значение в DEPTNO» даст следующее объединение строк:

	EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
[
	7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	300	
	7521	WARD	SALESMAN	7698	22-FEB-81	1250	500	
	7654	MARTIN	SALESMAN	7698	28-SEP-81	1250	1400	
	7698	BLAKE	MANAGER	7839	01-MAY-81	2850		
	7844	TURNER	SALESMAN	7698	08-SEP-81	1500	0]	30
[
	7782	CLARK	MANAGER	7839	09-JUN-81	2450		
	7839	KING	PRESIDENT		17-NOV-81	5000		
	7934	MILLER	CLERK	7782	23-JAN-82	1300]	10
[
	7566	JONES	MANAGER	7839	02-APR-81	2975		
	7788	SCOTT	ANALYST	7566	19-APR-87	3000		
	7876	ADAMS	CLERK	7788	23-MAY-87	1100		
	7902	FORD	ANALYST	7566	03-DEC-81	3000]	20

После группировки в окончательный ответ смогут поступить только значения признака группирования (данные из столбца DEPTNO, конкретно — значения 30, 10 и 20 для каждой группы) и агрегаты для групп, то есть то, что для каждой группы строк является общим. Это касается следующей фразы HAVING и всех далее идущих.

6.9.1.3. Промежуточный результат после HAVING

Присутствующая здесь фраза HAVING отсеет группы, количество сотрудников в которых больше четырех:

	EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
[
	7782	CLARK	MANAGER	7839	09-JUN-81	2450		
	7839	KING	PRESIDENT		17-NOV-81	5000		
	7934	MILLER	CLERK	7782	23-JAN-82	1300]	10
[
	7566	JONES	MANAGER	7839	02-APR-81	2975		
	7788	SCOTT	ANALYST	7566	19-APR-87	3000		
	7876	ADAMS	CLERK	7788	23-MAY-87	1100		
	7902	FORD	ANALYST	7566	03-DEC-81	3000]	20

Агрегаты (функции обобщения, в нашем случае это COUNT (*)) в запросах с GROUP BY автоматически приобретают необычный смысл: они распространяются только на группы, а не на все множество строк.

6.9.1.4. Промежуточный результат ORDER BY

Множество групп впервые упорядочивается — по убыванию значения DEPTNO:

	EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
[7876	ADAMS	CLERK	7788	23-MAY-87	1100		
	7566	JONES	MANAGER	7839	02-APR-81	2975		
	7788	SCOTT	ANALYST	7566	19-APR-87	3000		
	7902	FORD	ANALYST	7566	03-DEC-81	3000]	20
[7934	MILLER	CLERK	7782	23-JAN-82	1300		
	7782	CLARK	MANAGER	7839	09-JUN-81	2450		
	7839	KING	PRESIDENT		17-NOV-81	5000]	10

6.9.1.5. Конечный отбор SELECT

В окончательный ответ попадают только данные о номерах отделов:

```

DEPTNO
-----
      20
      10

```

В другом случае в ответ могли бы поступить еще и агрегаты.

6.9.2. Отсутствие значения в выражении для группировки

Подобно случаю с DISTINCT, отсутствующие значения в выражении, используемом для группировки, считаются *равными друг другу*:

```
SQL> SELECT comm, COUNT ( * ) FROM emp GROUP BY comm;
```

```

      COMM      COUNT ( * )
-----
          0              1
         300              1
         500              1
        1400              1
                10

```

Причина та же: иначе запросы с группировкой были бы малопрактичны. Можно напомнить, что это исключение для правила сравнения с NULL касается внутренней технической операции сравнения, но не явно сформулированной в запросе.

6.9.3. Другие примеры

Выдать информацию о сотрудниках в отделах, где они есть, перечислив сотрудников в порядке поступления на работу:

```

COLUMN names FORMAT A45
SELECT
  deptno
, MIN ( hiredate ) first
, MAX ( hiredate ) last
, LISTAGG ( ename, ' , ' ) WITHIN GROUP ( ORDER BY hiredate ) names
FROM   emp
GROUP BY deptno
ORDER BY deptno
;
```

Ответ:

DEPTNO	FIRST	LAST	NAMES
10	09-JUN-81	23-JAN-82	CLARK, KING, MILLER
20	17-DEC-80	23-MAY-87	SMITH, JONES, FORD, SCOTT, ADAMS
30	20-FEB-81	03-DEC-81	ALLEN, WARD, BLAKE, TURNER, MARTIN, JAMES

В следующих примерах для группирования строк указывается пара значений, а не одно.

Сравните выдачу:

```
SELECT mgr, job, SUM ( sal ), COUNT ( * )
FROM emp
GROUP BY mgr, job
;
```

и:

```
SELECT mgr, job, SUM ( sal ), COUNT ( * )
FROM emp
GROUP BY job, mgr
;
```

Очевидно, порядок перечисления выражений для признаков группировки в GROUP BY не сказывается на конечном результате.

Еще пример:

```
SELECT mgr, job, COUNT ( * )
FROM emp
GROUP BY mgr, job
HAVING AVG ( sal ) > 1000
;
```

Дополнительные примеры приводятся в заключительной части настоящего материала.

Обратите внимание, что формально запрос с группировкой может и не содержать обращений к функциям обобщения (агрегатам), однако в этом случае проще отказаться от GROUP BY и использовать DISTINCT.

6.9.4. Указание ROLLUP, CUBE и GROUPING SETS в во фразе GROUP BY

В версии 8.1 Oracle в запросах с GROUP BY в соответствии с предложениями SQL:1999 (но без строгого им следования) была введена возможность заключать признаки группировки в конструкции ROLLUP и CUBE, позволяющие симитировать в СУБД на основе SQL функциональность OLAP. В версии 9 к ним добавилась конструкция GROUPING SETS.

Всякий запрос с ROLLUP, CUBE или GROUPING SETS может быть переформулирован в запрос с группировками, но *без* этих конструкций, и тем не менее их использование (а) дает более лаконичную и универсальную формулировку, и (б) специальным образом в Oracle оптимизировано.

Пример группировки с ROLLUP:

```
SELECT mgr, job, SUM ( sal ), COUNT ( * )
FROM emp
GROUP BY ROLLUP ( mgr, job )
;
```

Результат складывается из группировки по паре значений (MGR, JOB) плюс группировки по значению MGR плюс данных по «собственной группе» из всего множества исходных строк:

MGR	JOB	SUM (SAL)	COUNT (*)
-----	-----	-----------	-------------

PRESIDENT	5000	1	← SUM и COUNT для mgr=NULL и job=PRESIDENT
	5000	1	← SUM и COUNT для mgr=NULL
7566 ANALYST	6000	2	← SUM и COUNT для mgr=7566 и job=ANALYST
7566	6000	2	← SUM и COUNT для mgr=7566
7698 CLERK	950	1	← SUM и COUNT для mgr=7698 и job=CLERK
7698 SALESMAN	5600	4	← SUM и COUNT для mgr=7698 и job=SALESMAN
7698	6550	5	← SUM и COUNT для mgr=7698
7782 CLERK	1300	1	← ...
7782	1300	1	
7788 CLERK	1100	1	
7788	1100	1	
7839 MANAGER	8275	3	
7839	8275	3	
7902 CLERK	800	1	
7902	800	1	
	29025	14	← SUM и COUNT для всех начальников и должностей

Такое построение результата искусственно «порождает» в ответе отсутствующие значения из-за группировок по сокращенным перечням признаков. Обратите внимание: они имеют смысл «значение неприменимо» (unapplicable). Это противоречит упоминавшейся выше рекомендации стандарта SQL использовать NULL в смысле «значение неизвестно» (unknown). Такую специфику отсутствующих значений в результатах с GROUP BY ROLLUP следует учитывать в последующей обработке. Но ведь пропуски значений могли иметься и в исходных данных (например, у сотрудника могла быть не обозначена должность), где они могут иметь иной смысл и требовать иного характера обработки. Программа, получив от СУБД ответ, не в состоянии определить смысл NULL в каждом конкретном случае только на основании формулировки запроса.

Для различения отсутствующих значений в строках с обобщенными итогами от значений, отсутствующих в исходных данных, специально предусмотрена индикаторная функция GROUPING, возвращающая 1 на «благоприобретенных» NULL-пропусках и 0 на всех остальных. Следующий запрос получен из предшествующего добавлением обращений к функции GROUPING с целью пояснить, как она работает:

```
SELECT
    GROUPING ( mgr )
,   GROUPING ( job )
,   mgr
,   job
,   SUM ( sal )
,   COUNT ( * )
FROM
    emp
GROUP BY
    ROLLUP ( mgr, job )
;
```

С помощью функции GROUPING и оператора CASE отсутствующие значения в строках промежуточных сумм могут быть заполнены желаемым образом:

```
COLUMN "Начальник" FORMAT A15
SELECT
    CASE GROUPING ( mgr ) WHEN 1 THEN 'все начальники' ELSE TO_CHAR ( mgr ) END
    AS "Начальник"
,   CASE GROUPING ( job ) WHEN 1 THEN 'все должности' ELSE job END
    AS "Должность"
,   SUM ( sal ) AS "Зарплата в группе"
,   COUNT ( * ) AS "Сотрудников в группе"
FROM
    emp
GROUP BY
```

```

ROLLUP ( mgr, job )
ORDER BY mgr, job
;

```

Упражнение. Выполнить приведенный запрос и посмотреть результат. Заменить ROLLUP на CUBE, выполнить и посмотреть результат.

В версии 9 вместо функции GROUPING стало возможным использовать функцию GROUPING_ID, возвращающую тот же результат, что и GROUPING, но в виде поразрядной маски. Так, один из предшествующих «пояснительных» запросов можно переписать короче:

```

SELECT
    GROUPING_ID ( mgr, job )
, mgr
, job
, SUM ( sal )
, COUNT ( * )
FROM
    emp
GROUP BY
    ROLLUP ( mgr, job )
;

```

Иногда использование GROUPING_ID позволяет проще, нежели GROUPING, сформулировать в запросе фильтр.

Еще одна функция, GROUP_ID, также введена в версии 9 и позволяет в запросе с группировкой отметить повторяющиеся строки с агрегатами (которые могут иногда возникать), указывая «степень повторения» > 0, например:

```

SELECT mgr, job, SUM ( sal ), COUNT ( * ), GROUP_ID ( )
FROM    emp
GROUP BY mgr, ROLLUP ( mgr, job )
;

```

В версии 9 появилась возможность выдавать в запросе *только промежуточные агрегаты* без усложнений текста, необходимых в таких ситуациях при использовании ROLLUP и CUBE. Для этого введено специальное указание GROUPING SETS:

```

SELECT mgr, job, SUM ( sal ), COUNT ( * )
FROM    emp
GROUP BY GROUPING SETS ( mgr, job )
;

```

Сравните с результатом запроса:

```

SELECT mgr, job, SUM ( sal ), COUNT ( * )
FROM    emp
GROUP BY GROUPING SETS ( mgr ), GROUPING SETS ( job )
;

```

6.9.5. Несамостоятельность группировки с обобщениями ROLLUP, CUBE и GROUPING SETS

Использование в группировках ROLLUP, CUBE и GROUPING SETS дает гибкость формулировки и эффективность исполнения, но не логическую новизну. Так, следующие запросы «по определению» равносильны по результату:

SELECT a, b FROM Z GROUP BY ROLLUP (a, b)	SELECT a, b FROM Z GROUP BY a, b UNION ALL SELECT a, NULL FROM Z GROUP BY a UNION ALL SELECT NULL, NULL FROM Z
---	--

SELECT a, b FROM Z GROUP BY CUBE (a, b)	SELECT a, b FROM Z GROUP BY a, b UNION ALL SELECT a, NULL FROM Z GROUP BY a UNION ALL SELECT NULL, b FROM Z GROUP BY b UNION ALL SELECT NULL, NULL FROM Z
SELECT a, b FROM Z GROUP BY GROUPING SETS (a, b)	SELECT a, NULL FROM Z GROUP BY a UNION ALL SELECT NULL, b FROM Z GROUP BY b

В списке выражений для ROLLUP, CUBE и GROUPING SETS также возможно группирование. Например, возможен запрос типа (AGG — условное обозначение произвольной агрегирующей функции):

```
SELECT a, b, c, AGG ( d )
FROM Z
GROUP BY ROLLUP ( a, ( b, c ) )
```

По аналогии, понять его смысл помогает следующая равносильная формулировка:

```
SELECT a, b, c, AGG ( d ) FROM Z GROUP BY a, b, c UNION ALL
SELECT a, NULL, NULL, AGG ( d ) FROM Z GROUP BY a UNION ALL
SELECT NULL, NULL, NULL, AGG ( d ) FROM Z
```

То есть при подсчете обобщений, пары (тройки, четверки, ...) выражений будут приниматься как единое целое («составной», composite, «столбец»). Группирование может быть произвольным: ((a, b), c), (a, (b, c), d), ((a, b), (c, d, e)) и так далее. Равным образом сказанное распространяется на CUBE и GROUPING SETS. Возможно также и комбинирование типа следующего:

```
SELECT a, b, c, AGG ( d )
FROM Z
GROUP BY GROUPING SETS ( a, ROLLUP ( b, c ) )
```

Это равносильно формулировке:

```
SELECT a, NULL, NULL, AGG ( d ) FROM Z GROUP BY a
UNION ALL
SELECT NULL, b, c, AGG ( d ) FROM Z GROUP BY ROLLUP ( b, c )
```

6.10. Фраза CONNECT BY предложения SELECT

Для моделирования в БД фрагментов прикладной области порой удобна иерархическая (многоуровневая) организация данных. Фраза CONNECT BY используется для запросов по иерархически организованным записям. Она представляет собой специфичное расширение SQL в Oracle, несовместимое с другими диалектами SQL (применяющих с той же целью собственные конструкции) и отсутствующее в стандарте.

Элементами фразы CONNECT BY являются:

- указание строк, для которых следует вывести предков или потомков в иерархии (конструкция START WITH);
- ссылка на поля предшествующей записи на очередном шаге рекурсии (ключевое слово PRIOR), определяющая на деле продвижение в иерархии вверх, либо вниз;
- условное выражение, формулирующее иерархическую, по мнению программиста, связь среди строк.

Условное выражение, задающее иерархию, программист всякий раз вынужден указывать в запросе явочным порядком, так как ни SQL, ни тем более реляционная модель, не дают средств определять иерархическую зависимость между строками в таблице (в отношении). Это не случайно, и вызвано тем обстоятельством, что изначально SQL создавался как «язык логики первого порядка», где выразительных средств для задания рекурсии, предполагаемой иерархией, не предусмотрено. При желании проектировщика БД иметь в данных иерархию, это обстоятельство весьма неприятно, и приводит к тому, что подобная крайне ответственная часть описания данных не хранится в БД, и вообще не хранится, иначе как в голове у программиста.

В то же время начальное множество строк (конструкция START WITH), и даже ссылка на поля предшествующей при рекурсивной обработке записи (указание PRIOR) могут отсутствовать.

Общий алгоритм вычислений, выполняемых фразой CONNECT BY, можно условно описать следующим образом:

Начальные_строки := { *Входное_множество* | строки, обозначенные в **START WITH** };
Результат_CONNECT_BY := *Начальные_строки*;

для каждой *Строки* **из** *Начальных_строк* **выполнить:**

Рекурсия (*Поля_PRIOR* **= пусто):**

для каждой *Новой_Строки* **из** *Входного_множества*

если *Условное_выражение (Поля_PRIOR, Новая_Строка)* = TRUE **то**

Результат_CONNECT_BY := *Новая_Строка*;

Рекурсия (*Поля_PRIOR* **из** *Новой_Строки* **);**

конец цикла;

конец рекурсии;

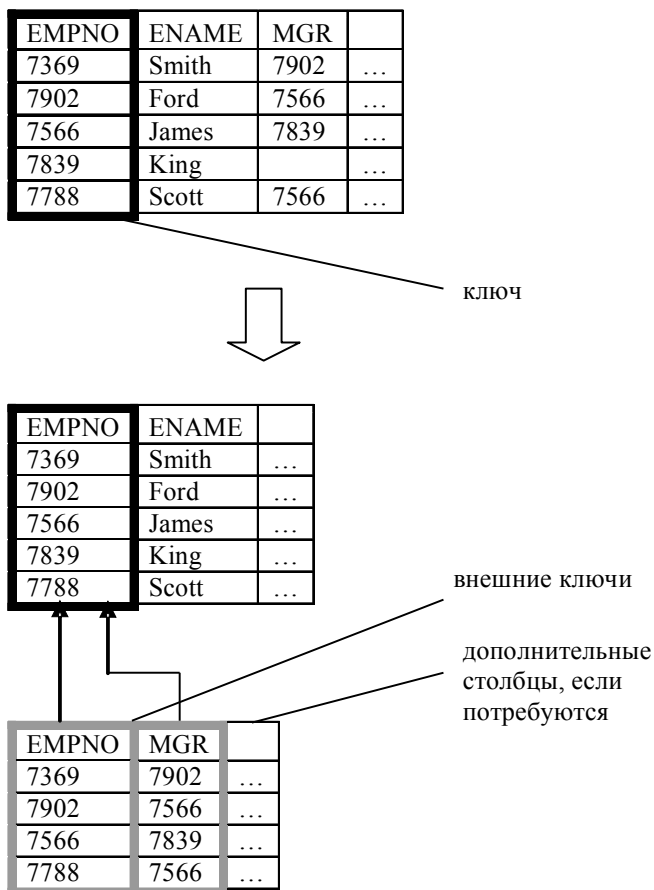
Из алгоритма следует, что строки ответа поступают от СУБД в порядке прохода иерархии, а не случайно. Если в будущем фирма Oracle внесет изменения в алгоритм вычислений, то порядок строк в результате может стать иным, чем ранее. В то же время в ближайшей перспективе этого не произойдет, и в отдаленной маловероятно. Полагаться ли на неявный порядок результата, или же нет, программист волен решать сам. В любом случае явное средство задания порядка строк-результата в запросах с CONNECT BY в Oracle тоже имеется, и будет рассмотрено ниже.

Другое следствие из приведенного алгоритма — это то, что в множество как предков, так и потомков, по определению включены данные из начальных строк. По большому счету это условность, и от такого включения данных начальных строк можно было бы отказаться, однако в принятом решении есть свои положительные моменты.

6.10.1. Хранение древовидно зависимых данных в БД

Наиболее востребованным видом иерархии является дерево. Древовидная зависимость сотрудников в таблице EMP представлена столбцами EMPNO (уникальный «табельный номер сотрудника») и MGR («табельный номер руководителя сотрудника»). Это именно тот способ задания древовидной зависимости данных, который чаще всего встречается в жизни (классификаторы; устройство сложных агрегатов на производстве; структура организации и так далее).

В жизни однако, этот способ часто применяется с той поправкой, что ссылка на предка устраняется из основной таблицы, а зависимость «предок — потомок» выносится в особую самостоятельную таблицу:



Достоинств у такого решения два:

- (а) столбец MGR теперь полностью заполнен и не имеет (единственного !) пропущенного значения;
- (б) в отдельной таблице, описывающей подчиненность сотрудников, открылась возможность естественным образом указать дополнительные свойства такой подчиненности, как, например, время работы под началом конкретного руководителя или что-нибудь еще.

Есть и недостаток: запрос о сотрудниках теперь придется обращаться уже к двум таблицам, и это более затратно. С точки зрения создателей реляционной модели ответственность за это несут разработчики СУБД.

6.10.2. Примеры запросов по дереву

Пример употребления фразы CONNECT BY в запросе о сотрудниках:

```
SELECT      ename
FROM        emp
CONNECT BY PRIOR empno = mgr
START WITH ename = 'SCOTT'
;
```

Будут выданы все подчиненные сотрудника SCOTT. Пример показывает использование слова PRIOR во фразе CONNECT BY, а также конструкции START WITH. Слово PRIOR (оператор, «системная функция») приписывается ведущему *столбцу* в отношении упорядочения, а не *выражению*, в котором упоминается столбец, то есть, указывается в сочетании «PRIOR *имя_столбца*». Выбор программистом ведущего столбца из пары, участвующей в построении условного выражения, фактически задает направление движения по дереву: поиск родителей или же потомков (как в данном случае).

Ведущий столбец можно употребить и во фразах, следующих за фразой CONNECT BY в логической схеме обработки предложения SELECT; сравните предыдущий пример со следующим:

```
SELECT      ename, PRIOR ename
FROM        emp
CONNECT BY  PRIOR empno = mgr
START WITH  ename = 'SCOTT'
ORDER BY    PRIOR ename
;
```

Степень отдаленности от начального узла просмотра дерева показывает специальная системная функция без параметров LEVEL («псевдостолбец» по терминологии Oracle), доступная исключительно в запросах с CONNECT BY:

```
SELECT      LEVEL, ename
FROM        emp
CONNECT BY  empno = PRIOR mgr
START WITH  ename IN ( 'SCOTT', 'ALLEN' )
;
```

(Все начальники сотрудников SCOTT и ALLEN с указанием уровня подчиненности).

Если не указать START WITH, исходное множество строк будет составлено из всех имеющихся:

```
SELECT      LEVEL, ename
FROM        emp
CONNECT BY  PRIOR empno = mgr
;
```

(«Лес» деревьев подчиненности всех сотрудников друг другу.)

Практически сегодня запросы с использованием LEVEL не очень интересны ввиду имеющихся с версии 9 более удобных для употребления системных функций.

Упражнение. Выдать дерево подчиненности сотрудников, используя для удобного представления на экране отступы из пробелов. Для формирования отступов подойдут функции RPAD и LPAD.

Формально в выражении правила иерархии указывать PRIOR не обязательно, однако в этом случае запрос по технике исполнения останется рекурсивным, но уже не будет иерархичен. Тем не менее, в технических целях такое построение запроса иногда находит себе применение, например как способ получить источник данных, составленный из произвольного количества строк:

```
SELECT      LEVEL
FROM        dual
CONNECT BY  LEVEL <= 10
;
```

Это выглядит довольно эффектно. Составление логического выражения, однако, здесь требует повышенного внимания, чтобы порождение строк не вылилось бы в «бесконечную» работу. (На деле произвольность количества строк, и даже корректность исполнения подобного запроса могут оказаться негарантированными в силу внутренних технических особенностей исполнения. Например, в версиях Oracle до 10.2 этот запрос мог при любых обстоятельствах возвращать не более 25 строк, а с версии 10.2 при особо больших *n* может возникнуть ошибка выполнения ORA-30009, вызванная нехваткой рабочей памяти сеанса, и причем тем раньше, чем меньше имеется доступной памяти сеанса.)

В последнем запросе вместо LEVEL с равным успехом и тем же результатом можно сослаться на ROWNUM.

Если последний запрос слегка изменить, он будет выдавать столбец с ровно одним пропущенным значением:

```

SELECT      LEVEL - ( PRIOR 1 )
FROM        dual
CONNECT BY LEVEL <= 10
;

```

Это вызвано тем, что на первой выданной строке выражение PRIOR *все_что_угодно* не вернет ничего (NULL). Но такой способ достичь подобный результат, конечно, не является единственным.

6.10.3. Фильтрация узлов дерева

При необходимости какие-то узлы дерева в выдачу можно не включать. Фильтрацию можно вставить во фразу CONNECT BY или во фразу WHERE. Так как логически CONNECT BY обрабатывается ранее WHERE (см. логическую схему обработки предложения SELECT выше), результаты и смысл фильтрации в этих случаях будут разными.

Пример исключения из списка подчиненных сотрудника SCOTT и всех его потомков:

```

SELECT      ename, PRIOR ename
FROM        emp
CONNECT BY PRIOR empno = mgr AND ename <> 'SCOTT'
      START WITH mgr IS NULL
;

```

Пример исключения из списка всех потомков сотрудника SCOTT:

```

SELECT      ename, PRIOR ename
FROM        emp
WHERE       ename <> 'SCOTT'
CONNECT BY PRIOR empno = mgr
      START WITH mgr IS NULL
;

```

Заметьте, что в тексте WHERE предшествует CONNECT BY, но это не соответствует порядку обработки.

Уточнить смысл приведенных двух запросов помогает рекурсивный алгоритм обработки CONNECT BY, приведенный ранее.

6.10.4. Специальные системные функции в предложениях с CONNECT BY

Специально для запросов по дереву создан ряд системных функций. Один пример — LEVEL — уже приводился. Другой пример — функция SYS_CONNECT_BY_PATH, позволяющая получить для каждой строки полный ее «путь», считая от точки отсчета:

```

COLUMN epath FORMAT A100

SELECT      LEVEL, SYS_CONNECT_BY_PATH ( ename, '/' ) epath
FROM        emp
CONNECT BY PRIOR empno = mgr
      START WITH mgr IS NULL
;

```

Список функций, специально предназначенных для употребления в предложениях с CONNECT BY:

Функция	Описание
LEVEL	Номер уровня в дереве (1 — корень, 2 — нижележащий уровень и т. д.)
SYS_CONNECT_BY_PATH (столбец, разделитель) ⁽⁹⁻⁾	Путь от корня дерева к узлу

CONNECT_BY_ISCYCLE ^[10-]	1, если потомок узла является одновременно его предком, иначе 0
CONNECT_BY_ISLEAF ^[10-]	1, если узел не имеет потомков
CONNECT_BY_ROOT (столбец) или CONNECT_BY_ROOT столбец ^[10-]	Значение из строки-корня
PRIOR (столбец) или PRIOR столбец	Значение из строки-прямого предка

^[9-] начиная с версии 9

^[10-] начиная с версии 10

6.10.5. Упорядочение результата

Фраза CONNECT BY выдает в результате дерево, но не заботится о порядке перечисления «веток» в пределах одного уровня. Упорядочить по заданному критерию ветви дерева традиционными средствами возможно, но делать это крайне неудобно. С версии 9 задачу много проще решить употреблением специальной фразы ORDER SIBLINGS BY:

```
SELECT          LEVEL, SYS_CONNECT_BY_PATH ( ename, '/' ) epath
FROM            emp
CONNECT BY      PRIOR empno = mgr
START WITH      mgr IS NULL
ORDER SIBLINGS BY ename
;
```

Упражнение. Получите от Oracle ответ на последний запрос. Замените в тексте ORDER SIBLINGS BY ename на (1) ORDER BY ename, на (2) ORDER BY epath; задайте видоизмененные запросы. Сравните результаты и объясните разницу.

Фразы ORDER BY и ORDER SIBLINGS BY в предложении SELECT взаимоисключающие.

6.10.6. Обработка заикливания

Поскольку Oracle не контролирует корректность иерархической взаимосвязи хранимых строк (БД попросту «не понимает» такой взаимосвязи), ответственность за ее соблюдение ложится на программиста. При изменении данных в БД он может нарушить взаимосвязь, случайно или намеренно. Если такое нарушение приводит к заикливанию, рекурсивно исполняющаяся фраза CONNECT BY обнаружит это и выдаст ошибку:

```
SQL> UPDATE emp SET mgr = 7876 WHERE ename = 'JONES';

1 row updated.
```

У Джонса начальником поставлен Адамс (теперь Адамс → Джонс; здесь стрелка указывает на подчиненного), но тот же Адамс и среди его подчиненных (Джонс → Скотт → Адамс):

```
SQL> SELECT          SYS_CONNECT_BY_PATH ( ename, '/' ) epath
2 FROM            emp
3 CONNECT BY      PRIOR empno = mgr
4 START WITH      ename = 'JONES'
5 /
```

ERROR:

ORA-01436: CONNECT BY loop in user data

Указание NOCYCLE заставит Oracle завершить рекурсивный просмотр записей при обнаружении заикленности и не сообщать об ошибке:

```
SQL> SELECT          SYS_CONNECT_BY_PATH ( ename, '/' ) epath
```

```

2 FROM      emp
3 CONNECT BY NOCYCLE PRIOR empno = mgr
4* START WITH ename = 'JONES'
5 /

```

EPATH

```

-----
/JONES
/JONES/SCOTT
/JONES/SCOTT/ADAMS
/JONES/FORD
/JONES/FORD/SMITH

```

SQL> ROLLBACK;

В любом случае «бесконечного» выполнения запроса при использовании CONNECT BY не случится.

6.10.7. Недревовидная иерархия

Фраза CONNECT BY способна рекурсивно обрабатывать не только древовидно организованные данные, но и иерархию общего вида. Выполним:

```

CREATE TABLE ways (
  node      VARCHAR2 ( 20 )
, parent    VARCHAR2 ( 20 )
, distance  NUMBER   ( 5 )
);
INSERT INTO ways VALUES ( 'Ленинград', 'Москва',      696 );
INSERT INTO ways VALUES ( 'Новгород',  'Москва',      538 );
INSERT INTO ways VALUES ( 'Ленинград', 'Новгород',    179 );
INSERT INTO ways VALUES ( 'Выборг',    'Ленинград',    135 );
COMMIT;

```

Обратите внимание, что создана не «таблица с расстояниями», а таблица с направленными маршрутами, предоставляющая расстояния между городами с точки зрения Москвы (здесь — единственная вершина иерархии). Такое представление данных и приводимые ниже запросы плохо подходят для решения более общей задачи поиска маршрута между двумя произвольными точками.

Запрос вниз по иерархии от узла 'Москва' (присутствует только в качестве предка):

```

SQL> COLUMN route FORMAT a40
SQL> SELECT      SYS_CONNECT_BY_PATH ( node, '/' ) route
2 FROM          ways
3 CONNECT BY PRIOR node = parent
4 START WITH parent = 'Москва'
5 ;

```

ROUTE

```

-----
/Ленинград
/Ленинград/Выборг
/Новгород
/Новгород/Ленинград
/Новгород/Ленинград/Выборг

```

Запрос вверх по иерархии от узла 'Выборг':

```

SQL> SELECT      SYS_CONNECT_BY_PATH ( node, '/' ) route
2 FROM          ways
3 CONNECT BY node = PRIOR parent
4 START WITH node = 'Выборг'
5 ;

```

ROUTE

/Выборг
/Выборг/Ленинград
/Выборг/Ленинград
/Выборг/Ленинград/Новгород

Упражнение. Внести в таблицу WAYS заикленность данных и проверить реакцию фразы CONNECT BY на цикл.

6.11. Комбинирование результатов SELECT множественными операциями

Результатами вычисления предложений SELECT являются множества строк. SQL позволяет применять для таких множеств три классические в математике операции объединения, пересечения и вычитания. Эти операции (с некоторыми искажениями) унаследованы от реляционного подхода, где они определены на отношениях.

6.11.1. Сложение строк — результатов SELECT оператором UNION

Объединение результатов двух или более запросов.

«Выдать номера сотрудников, имеющих должность MANAGER или имеющих подчиненных»:

```
SELECT mgr FROM emp WHERE mgr IS NOT NULL
UNION
SELECT empno FROM emp WHERE job = 'MANAGER'
;
```

- UNION автоматически убирает дубликаты (повторения) строк из результата.
- Устранение дубликатов требует внутренней работы СУБД, отчего на больших объемах данных программа получит результат нескоро.

Разновидность UNION **ALL** будет отрабатываться как простое объединение строк *без* устранения дубликатов. «Выдать номера сотрудников, имеющих должность MANAGER, и номера сотрудников, имеющих подчиненных»:

```
SELECT mgr FROM emp WHERE mgr IS NOT NULL
UNION ALL
SELECT empno FROM emp WHERE job = 'MANAGER'
;
```

Если программист, зная свои данные, уверен, что строки в объединении результатов запросов не повторяются, ему следует использовать именно UNION ALL и существенно сэкономить на этом ресурсы СУБД, включая процессорное время обработки.

6.11.2. Пересечение результатов SELECT оператором INTERSECT

Пересечение результатов двух или более запросов достигается оператором INTERSECT.

Пример. «Выдать номера сотрудников, имеющих должность MANAGER и имеющих подчиненных»:

```
SELECT mgr FROM emp WHERE mgr IS NOT NULL
INTERSECT
SELECT empno FROM emp WHERE job = 'MANAGER'
;
```

6.11.3. Вычитание результатов SELECT оператором MINUS

Вычитание результата одного запроса из другого достигается использованием оператора MINUS.

Пример. «Выдать номера сотрудников, имеющих подчиненных, но не в должности MANAGER»:

```
SELECT mgr FROM emp WHERE mgr IS NOT NULL
MINUS
SELECT empno FROM emp WHERE job = 'MANAGER'
;
```

Упражнение. Поменяйте в последнем запросе предложения SELECT местами: выдайте список «сотрудников в должности MANAGER, но не имеющих подчиненных». Пустой ответ означает, что таких нет.

Упражнение. Выдать с помощью множественной операции MINUS названия отделов, где нет сотрудников.

В Oracle имя операции MINUS взято из реляционной модели, а в стандарте SQL:1999 соответствующий оператор называется EXCEPT (последнего названия придерживаются, например, DB2 и SQL Server).

6.11.4. Общие правила

- Комбинируемые блоки SELECT должны иметь одинаковую структуру (в Oracle — с точностью до *совместимости* типов столбцов: общего формата хранения, но не обязательно точности).
- При комбинировании запросов автоматически убираются дубликаты (за исключением UNION ALL).
- При комбинировании запросов отсутствующие значения (NULL) считаются *равными* друг другу (исключение из общего правила сравнения с отсутствующим значением) подобно тому, как это происходит при использовании DISTINCT или GROUP BY.
- Фраза ORDER BY может следовать только за последним SELECT и применяться к общему результату.
- Именование столбцов окончательного результата по правилам первого предложения SELECT.

Нехарактерное для SQL самопроизвольное устранение повторяющихся строк при выполнении UNION, INTERSECT и MINUS требует внимания программиста. Например, в случае наличия повторений в столбцах-операндах UNION нельзя заменить на однопроходный SELECT с условным выражением через OR, а INTERSECT — с условным выражением через AND. Следующие два предложения дадут разные ответы:

```
SELECT mgr FROM emp WHERE deptno = 10 AND mgr IS NOT NULL
UNION
SELECT mgr FROM emp WHERE deptno = 20 AND mgr IS NOT NULL
;

SELECT mgr FROM emp
WHERE ( deptno = 10 OR deptno = 20 ) AND mgr IS NOT NULL
;
```

Приравнивание отсутствующих значений при внутреннем сравнении («NULL = NULL») также не позволяет расслабляться. Рассмотрим запрос:

```
SELECT job, comm FROM emp WHERE deptno = 30
UNION
SELECT job, comm FROM emp WHERE deptno = 10
;
```

Обратите внимание, что в этих отделах есть по два менеджера и клерка с отсутствующими коммиссионными, но в ответе это обстоятельство не было учтено.

Пример явного именованя столбцов и упорядочения строк окончательного ответа:

```
SELECT mgr "Начальники не менеджеры" FROM emp WHERE mgr IS NOT NULL
MINUS
SELECT empno FROM emp WHERE job = 'MANAGER'
ORDER BY "Начальники не менеджеры"
;
```

6.12. Операция соединения в предложении SELECT

Запрос SELECT считается соединением, если обращается к нескольким источникам данных при том, что столбцы разных источников сравниваются друг с другом (в общем случае условным выражением общего вида). Соединение является одной из основных операций в реляционной модели, а в SQL она попала в искаженном виде по причине необязательности в SQL правила ключа применительно к таблицам.

Операция соединения издавна привлекала внимание разработчиков СУБД, так как, с одной стороны, она неизбежно возникает в приложении вследствие нормализации отношений/таблиц БД (обоснованной теоретически), а с другой, излишне прямолинейная схема отработки соединения приводит к большими затратами СУБД.

По этим причинам операция соединения в свое время подробно исследовалась, подверглась систематизации и получила собственное синтаксическое оформление.

6.12.1. Виды соединений

Пример записи соединения таблиц в SQL:

```
SELECT emp.deptno, dept.dname
FROM emp, dept
WHERE emp.deptno = dept.deptno
;
```

Возможные варианты соотношений данных в соединяемых столбцах:

- столбцы заполнены одинаково;
- значения в одном столбце составляют подмножество значений в другом;
- множества значений в обоих столбцах пересекаются друг с другом;
- множества значений в обоих столбцах не пересекаются между собой.

Примеры и пояснения некоторых видов соединений, как то: тетасоединения, эквисоединения, естественного соединения, полуоткрытого и открытого, а также антисоединения приводятся ниже.

6.12.1.1. Тетасоединение

Примерный вид:

```
SELECT *
FROM emp, dept
WHERE emp.deptno <оператор_сравнения> dept.deptno
```

(когда оператор сравнения произвольный).

6.12.1.2. Эквисоединение

Примерный вид:

```
SELECT *  
FROM   emp, dept  
WHERE  emp.deptno = dept.deptno  
;
```

(тетасоединение, когда оператор сравнения — равенство).

6.12.1.3. Естественное соединение

Примерный вид:

```
SELECT emp.*, dept.dname, dept.loc  
FROM   emp, dept  
WHERE  emp.deptno = dept.deptno  
;
```

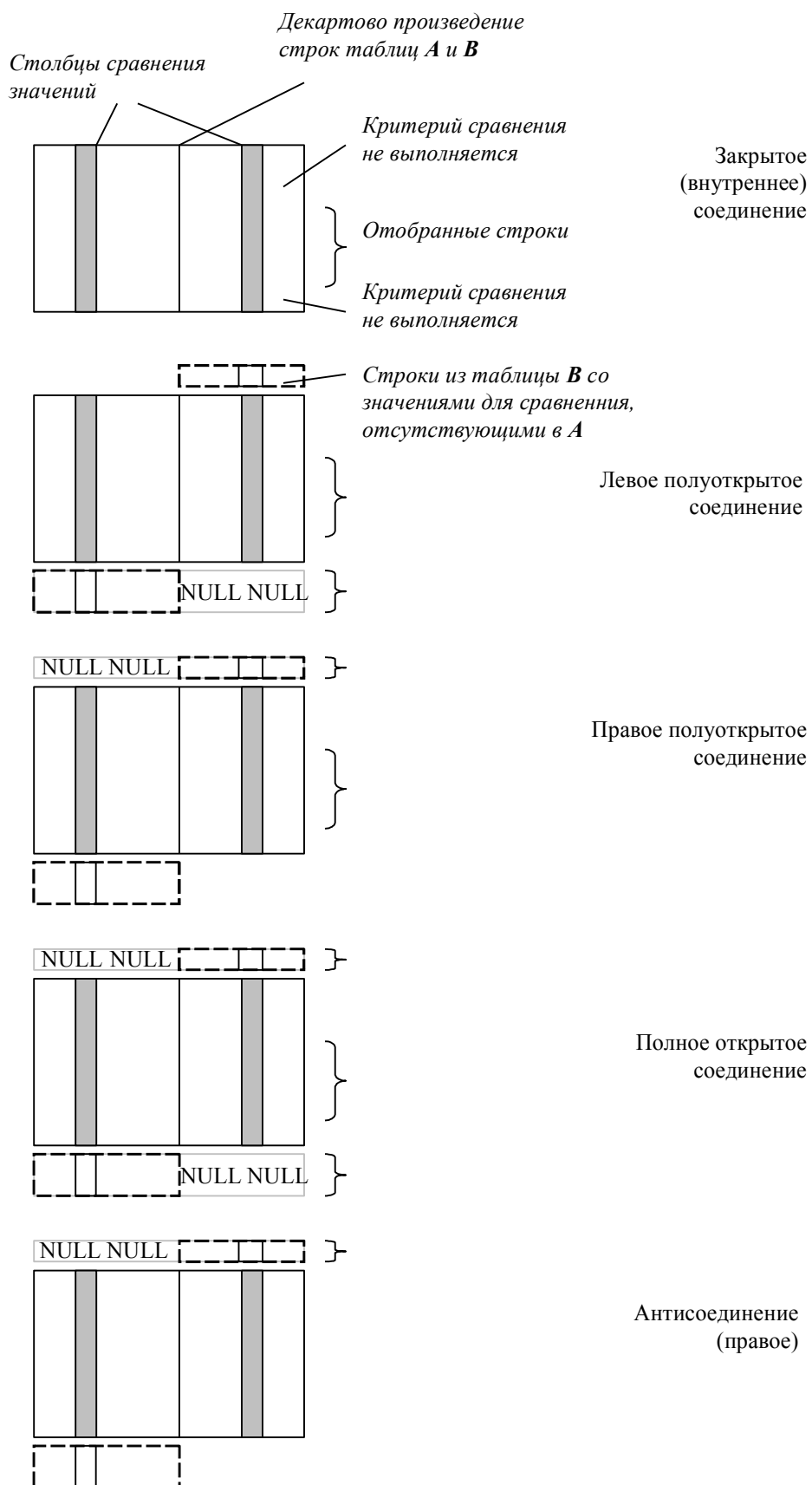
(когда оператор сравнения — равенство и соединяемые столбцы в таблицах именованы одинаково).

6.12.1.4. Полнота соединений

Имеются следующие виды соединений, диктующие разные схемы отбора строк в результат:

- закрытое
- полуоткрытое «левое»
- полуоткрытое «правое»
- открытое полное
- антисоединения «правое» и «левое»

На приводимом рисунке фигурными скобками обозначены строки, отбираемые для построения результата соединений перечисленных видов:



Из пар левое/правое полуоткрытые соединения и левое/правое антисоединения без ущерба функциональности можно было бы оставить по одной разновидности на выбор. Наличие именно пар избыточно, и обусловлено соображением удобства программирования.

Приводившийся выше пример естественного соединения является одновременно примером закрытого соединения, так как сравнение значений столбцов выполняется для всех строк с *присутствующими* значениями в паре (в более общем случае — в тройке, четверке и т.д.) столбцов.

Обратите внимание также, что полуоткрытые соединения порождают строки с NULL, при том что эти NULL имеют смысл «значение неприменимо». Это способно породить ту же проблему выбора способа дальнейшей обработки результата, что возникала в запросах с GROUP BY ROLLUP и CUBE. Однако здесь специальной функции-различителя нет, и смысл пропущенного значения должен определяться программистом на основе места его отсутствия.

6.12.1.5. Поясняющие примеры соединений

Здесь приводятся примеры разных видов соединений по критерию полноты. В данном случае речь идет о «самосоединениях», то есть соединениях по значениям столбцов фактически одной и той же таблицы-источника, указанной во фразе FROM более одного раза. Самосоединения позволяют взглянуть на одну таблицу с разных смысловых точек зрения. В нашем случае таблица EMP воспринимается (а) как таблица с данными о подчиненных, и (б) как таблица с данными о начальниках.

Пример закрытого соединения:

```
SELECT
    employees.ename, 'в подчинении у', managers.ename
FROM
    emp employees, emp managers
WHERE
    employees.mgr = managers.empno
;
```

В такой формулировке операция закрытого соединения не использует никакого особенного синтаксиса. Он требуется в соединениях других разновидностей и основан на приписывании одному из сравниваемых выражений обозначения '(+)'.

Пример полуоткрытого «влево» соединения:

```
SELECT
    employees.ename, 'в подчинении у', managers.ename
FROM
    emp employees, emp managers
WHERE
    employees.mgr = managers.empno ( + )
;
```

Пример «левого» антисоединения:

```
SELECT
    employees.ename, 'в подчинении у', managers.ename
FROM
    emp employees, emp managers
WHERE
    employees.mgr = managers.empno ( + ) AND managers.empno IS NULL
;
```

Пример полуоткрытого «право» соединения:

```
SELECT
    employees.ename, 'в подчинении у', managers.ename
```

```

FROM
    emp employees, emp managers
WHERE
    employees.mgr ( + ) = managers.empno
;

```

Пример «правого» антисоединения:

```

SELECT
    employees.ename, 'в подчинении у', managers.ename
FROM
    emp employees, emp managers
WHERE
    employees.mgr ( + ) = managers.empno AND employees.empno IS NULL
;

```

Упражнение. Проверить результат выполнения приведенных соединений.

До введения в версии Oracle 9 специального синтаксиса непосредственная запись полного, двустороннего открытого соединения была невозможна (иными словами приписать '(+)' к обоим соединяемым столбцам одновременно не разрешается), и ее приходилось моделировать объединением результатов двух полуоткрытых соединений.

Приведенные выше формулировки полуоткрытых и анти- соединений придают запросу некоторую краткость, но содержательно не сообщают нового диалекту SQL в Oracle (и формулировкам стандарта SQL, о которых речь далее). Здесь в очередной раз проявляется избыточность SQL и в Oracle, и в стандарте. Так, левое антисоединение дает возможность сформулировать себя без обращения к дополнительному синтаксису, например следующим образом:

```

SELECT employees.ename, 'в подчинении у', NULL ename
FROM   emp employees
WHERE  NOT EXISTS
      ( SELECT managers.ename
        FROM   emp managers
        WHERE  employees.mgr = managers.empno
      )
;

```

Такая запись выглядит более тяжеловесно, чем со специальным обозначением, зато смысл запроса стал более понятен. Еще «увесистее», но зато и более ясна в своем действии полученная из нее формулировка для левого полуоткрытого соединения:

```

SELECT employees.ename, 'в подчинении у', NULL
FROM   emp employees
WHERE  NOT EXISTS
      ( SELECT managers.ename
        FROM   emp managers
        WHERE  employees.mgr = managers.empno
      )
UNION ALL
SELECT employees.ename, 'в подчинении у', managers.empno
FROM   emp employees, emp managers
WHERE  employees.mgr = managers.empno
;

```

Аналогичная пара для правого анти- и открытого соединений может быть записана так:

```

SELECT NULL ename, 'в подчинении у', employees.ename
FROM   emp employees
WHERE  NOT EXISTS
      ( SELECT managers.ename
        FROM   emp managers

```

```

        WHERE employees.empno = managers.mgr
    )
;

SELECT NULL ename, 'в подчинении у', employees.ename
FROM emp employees
WHERE NOT EXISTS
    ( SELECT managers.ename
      FROM emp managers
      WHERE employees.empno = managers.mgr
    )
UNION ALL
SELECT managers.ename, 'в подчинении у', employees.ename
FROM emp employees, emp managers
WHERE employees.empno = managers.mgr
;

```

Приведенные примеры альтернативных формулировок не единственны. Примечательно, что часто Oracle технически обрабатывает разные формулировки по-разному (с разной эффективностью). В любом случае формулировка с '(+)' может рассматриваться всего лишь как условная запись, сообщающая тексту запроса краткость.

Упражнение. На основе приведенных формулировок построить запрос на полное самосоединение с информацией о подчиненности сотрудников.

6.12.2. Предостерегающий пример использования полуоткрытого соединения

Использование полуоткрытых соединений плодотворно для программиста, но их составление, как и многое в SQL, требует внимания. В этом и следующем разделах приводятся пример неумышленно неправильного составления запроса с полуоткрытым соединением и пример типового употребления.

Пусть требуется выдать список имен отделов, а также число работников и фонд зарплаты для каждого. Следующий запрос, казалось бы, позволяет это сделать:

```

SELECT  dname, COUNT ( * ) "Employees", SUM ( sal ) "Salary total"
FROM    emp, dept
WHERE   emp.deptno ( + ) = dept.deptno
GROUP BY dname
;

```

Однако судя по результату, запрос составлен неверно:

DNAME	Employees	Salary total
ACCOUNTING	3	8750
OPERATIONS	1	
RESEARCH	5	10875
SALES	6	9400

Для того, чтобы разобраться, в чем ошибка, можно вспомнить про логическую последовательность обработки запроса (в нашем случае она будет: FROM → WHERE → GROUP BY → SELECT) и уточнить промежуточный результат, сложившийся до вычисления GROUP BY, например:

```

SELECT  dname, ename
FROM    emp, dept
WHERE   emp.deptno ( + ) = dept.deptno
;

```

Упражнение. Предложите правильную формулировку запроса данных об отделах.

Примечание. Тот же запрос к существующим данным можно сформулировать и без употребления открытого соединения указанным выше способом, например следующим образом:

```
SELECT
  d.dname
, ( SELECT COUNT ( * ) FROM emp e WHERE e.deptno = d.deptno ) emp_count
, ( SELECT SUM ( sal ) FROM emp e WHERE e.deptno = d.deptno ) tot_sal
FROM dept d
;
```

Ответы СУБД, если и будут различаться, то никак не данными, а возможно только техникой обработки, однако тема оптимизации запросов здесь не рассматривается. Если ее не касаться, выбор конкретной формулировки запроса из нескольких допустимых в подобных случаях — дело вкуса программиста и удобства восприятия. Заметьте также, что в первом подзапросе последнего запроса с равным успехом можно указать как COUNT (*), так и, к примеру, COUNT (ename).

6.12.3. Типовой пример употребления полуоткрытого соединения

Ниже используется популярный случай употребления полуоткрытого соединения при построении отчетов. Предположим, нужно составить отчет о том, сколько сотрудников нанималось на работу за определенный период времени. Подготовим вспомогательную таблицу PIVOT_YEARS одним из возможных способов:

```
CREATE TABLE pivot_years AS
  SELECT      ( ROWNUM - 1 ) + 1980 AS year
  FROM        dual
  CONNECT BY  ROWNUM <= 10
;
```

Она плотно заполнена «значениями года», целыми числами от 1980 до 1989. Тогда следующий запрос выдаст сведения о количестве сотрудников, приходивших на работу в указанных в PIVOT_YEARS годах:

```
SELECT  p.year, COUNT ( e.empno )
FROM    pivot_years p, emp e
WHERE   p.year = EXTRACT ( YEAR FROM e.hiredate ( + ) )
GROUP BY p.year
ORDER BY p.year
;
```

Pivot table — это своего рода «реперная», или «опорная», «градуировочная», «калибровочная» таблица, помогающая анализировать данные (буквально — «разворотная» таблица). Ее можно сделать универсальной, если заполнить числами от 1 до *n*. Последний SELECT в этом случае придется слегка поправить.

Упражнение. Построить опорную таблицу со значениями от 1 до 10 и переформулировать основной запрос в расчете на нее.

Опорная таблица не обязана быть статичной. Аппарат табличных функций в PL/SQL позволяет построить функцию, способную порождать таблицу из *n* строк со значениями от 1 до *n* динамически. В SQL же ее несложно строить подзапросом с CONNECT BY или же рекурсивным (о котором речь ниже).

Упражнение. Использовать технику опорной таблицы для построения запроса о количестве подчиненных у всех имеющихся сотрудников.

Упражнение. Построить запрос для выяснения «не занятых» существующими сотрудниками табельных номеров в диапазоне [7900 .. 7935] (столбец EMPNO).

6.12.4. Синтаксис стандарта SQL:1999 для операции соединения

Приводимый ранее способ записи полуоткрытого соединения, хотя и не будучи придуманным фирмой СУБД Oracle, используется в ее СУБД, но не используется в стандартном SQL. В стандарт SQL:1999 для операции соединения были введены специально разработанные синтаксические конструкции. С версии 9 они включены в Oracle SQL. Они служат для оформления не только полуоткрытых, но и закрытых соединений (способных обходиться вообще без какого-нибудь особенного оформления), и рассматриваются ниже.

6.12.4.1. Закрытые соединения

Пример записи обычного (внутреннего, иначе — закрытого) соединения в соответствии с SQL:1999:

```
SELECT e.ename, d.dname
FROM   emp e
       INNER JOIN
       dept d
       ON ( e.deptno = d.deptno )
;
```

Обратите внимание, что такая формулировка разрешает привести в конструкции ON *любое* условное выражение (\leq , \geq , LIKE и так далее), например:

```
SELECT e.ename, e.sal, s.grade
FROM   emp e
       INNER JOIN
       salgrade s
       ON ( e.sal BETWEEN s.losal AND s.hisal )
;
```

Если же, как в предпоследнем случае, речь идет об *эквисоединении* (сравнении на совпадение величин) с одинаковыми названиями столбцов приравняемых значений, годится и иная формулировка:

```
SELECT e.ename, d.dname
FROM   emp e
       INNER JOIN
       dept d
       USING ( deptno )
;
```

У формулировки INNER JOIN ... USING есть отличие от формулировки INNER JOIN ... ON: во внутреннем декартовом произведении строк таблиц, которое строится фразой FROM в соответствии с логической схемой обработки предложения SELECT (приводилась выше) автоматически удаляются повторяющиеся столбцы. Это свойство унаследовано от реляционного соединения, где из результата автоматически удаляются одинаковые *атрибуты* (это не то же, что одинаково названные столбцы в таблицах SQL).

Упражнение. Сравните два результата, работы фразы FROM (в приводимых запросах кроме действий во фразе FROM по сути ничего не делается):

```
SELECT * FROM emp e INNER JOIN dept d ON e.deptno = d.deptno;

SELECT * FROM emp INNER JOIN dept USING ( deptno );
```

Как следствие соединения (вполне логичное) столбцы, участвующие в соединении, не требуют уточнения именем таблицы при ссылке на них в других местах предложения, например:

```
SELECT e.ename, d.dname, deptno
FROM   emp e
```

```

INNER JOIN
dept d
USING ( deptno )
;

```

С другой стороны, когда заходит речь о самосоединении, это может приводить к проблеме:

```

SQL> SELECT * FROM dept INNER JOIN dept USING ( deptno );
SELECT * FROM dept INNER JOIN dept USING ( deptno )
*
ERROR at line 1:
ORA-00918: column ambiguously defined

```

Успокаивает то, что с точки зрения приложения подобные запросы редко имеют смысл. Избавиться от этой ошибки Oracle можно введением псевдонимов. Следующие два запроса *не* приведут к ошибке:

```

SELECT * FROM dept a INNER JOIN dept b USING ( deptno );

SELECT *
FROM dept
INNER JOIN
( SELECT * FROM dept )
USING ( deptno )
;

```

Первый из них формально, а второй фактически не будет считаться самосоединением.

6.12.4.2. Два полуоткрытых и открытое соединения

Примеры (внешних) полуоткрытых соединений:

```

SELECT e.ename, d.dname
FROM emp e
LEFT OUTER JOIN
dept d
USING ( deptno )
;

SELECT e.ename, d.dname
FROM emp e
RIGHT OUTER JOIN
dept d
USING ( deptno )
;

```

Пример (внешнего) открытого соединения:

```

SELECT e.ename, d.dname
FROM emp e
FULL OUTER JOIN
dept d
USING ( deptno )
;

```

Последнее предложение не имеет равносильной записи в «старом» синтаксисе Oracle.

6.12.4.3. Естественные соединения

Для естественного закрытого соединения действует формулировка, подобная следующей:

```

SELECT e.ename, d.dname
FROM   emp e
       NATURAL INNER JOIN
       dept d
;

```

Фактически она работает как INNER JOIN ... USING по всем столбцам с совпадающими именами. Эта формулировка соблазнительна в силу своей простоты, однако в SQL она не поощряется некоторыми специалистами как упускающая контроль над фактическим набором столбцов соединения (ведь имена столбцов могут совпасть случайно и безотносительно к намерению разработчика БД служить средством соединения). А в реляционной модели такая формулировка соответствует единственно допустимой форме соединения и не имеет проблем потери контроля над способом соединения.

В то же время операция NATURAL INNER JOIN имеет относительную самостоятельность, хотя несколько превратно, но все же унаследованную от реляционной теории; ее можно использовать, если следить за именами столбцов соединяемых таблиц. Если столбцы таблиц вовсе *не* имеют совпадающих имен (в SQL !), то эта операция превращается в декартово произведение. Например, в таблице SALGRADE в схеме SCOTT три столбца: GRADE, LOSAL и HISAL и пять строк. Вот что даст естественное соединение этой таблицы с DEPT:

```
SQL> SELECT dname, grade FROM dept NATURAL INNER JOIN salgrade;
```

DNAME	GRADE
ACCOUNTING	1
ACCOUNTING	2
ACCOUNTING	3
ACCOUNTING	4
ACCOUNTING	5
RESEARCH	1
RESEARCH	2
RESEARCH	3
RESEARCH	4
RESEARCH	5
SALES	1
SALES	2
SALES	3
SALES	4
SALES	5
OPERATIONS	1
OPERATIONS	2
OPERATIONS	3
OPERATIONS	4
OPERATIONS	5

20 rows selected.

В таких случаях ответ будет совпадать с результатом действия другой операции, CROSS JOIN:

```
SELECT dname, grade FROM dept CROSS JOIN salgrade;
```

Если же наоборот, *все* столбцы естественно соединяемых таблиц совпадают, операция фактически превращается в пересечение строк таблиц, как в INTERSECT. Например:

```
SQL> SELECT dname, loc FROM dept a NATURAL INNER JOIN dept b;
```

DNAME	LOC
ACCOUNTING	NEW YORK
RESEARCH	DALLAS
SALES	CHICAGO
OPERATIONS	BOSTON

Обратите внимание на неочевидное обстоятельство. Если в последнем запросе отказаться от псевдонимов, отсева повторений не происходит; значения считаются разными:

```
SQL> SELECT dname, loc FROM dept NATURAL INNER JOIN dept;
```

DNAME	LOC
ACCOUNTING	NEW YORK
ACCOUNTING	NEW YORK
ACCOUNTING	NEW YORK
ACCOUNTING	NEW YORK
RESEARCH	DALLAS
RESEARCH	DALLAS
RESEARCH	DALLAS
RESEARCH	DALLAS
SALES	CHICAGO
SALES	CHICAGO
SALES	CHICAGO
SALES	CHICAGO
OPERATIONS	BOSTON
OPERATIONS	BOSTON
OPERATIONS	BOSTON
OPERATIONS	BOSTON

16 rows selected.

Заметьте, что особый эффект последней формулировки исчезает, когда соединяются две разные таблицы, пусть с одинаковой структурой:

```
SQL> CREATE TABLE dept1 AS SELECT * FROM dept;
```

Table created.

```
SQL> SELECT * FROM dept NATURAL INNER JOIN dept1;
```

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

Естественными могут быть и полуоткрытые соединения:

```
SELECT e.ename, d.dname
FROM   emp e
       NATURAL LEFT OUTER JOIN
       dept d
;
```

```
SELECT e.ename, d.dname
FROM   emp e
       NATURAL RIGHT OUTER JOIN
       dept d
;
```

Оговорки употребления, сделанные для естественного внутреннего соединения (лаконичность формулировки и риски человеческих ошибок), распространяются и на эти случаи.

6.12.4.4. Дополнительные примеры формулировок

Специальный синтаксис записи соединения не препятствует наличию в предложении SELECT фразы отбора строк WHERE:

```
SELECT e.ename, d.dname
FROM   emp e
       FULL OUTER JOIN
       dept d
       USING ( deptno )
WHERE  deptno <> 10
       AND d.dname <> 'RESEARCH'
;
```

Пример многократного (здесь — двойного) соединения:

```
SELECT m.ename, d.dname, d.loc
FROM   emp m
       INNER JOIN
       ( emp e
         INNER JOIN
         dept d
         USING ( deptno )
       )
       ON ( e.empno = m.mgr )
;
SELECT e.ename, d.dname, e.sal, s.grade
FROM   ( emp e
         INNER JOIN
         salgrade s
         ON e.sal BETWEEN s.losal AND s.hisal
       )
       INNER JOIN dept d
       ON e.deptno = d.deptno
;
```

Последнее равносильно запросу:

```
SELECT m.ename, d.dname, d.loc
FROM   emp m
       INNER JOIN
       emp e
       INNER JOIN
       dept d
       USING ( deptno )
       ON ( e.empno = m.mgr )
;
SELECT e.ename, d.dname, e.sal, s.grade
FROM   emp e
       INNER JOIN
       salgrade s
       ON e.sal BETWEEN s.losal AND s.hisal
       INNER JOIN dept d
       ON e.deptno = d.deptno
;
```

Следующее оформление того же запроса предлагает SQL Developer своим встроенным средством автоматического форматирования, полагая, очевидно, найти понимание у программистов:

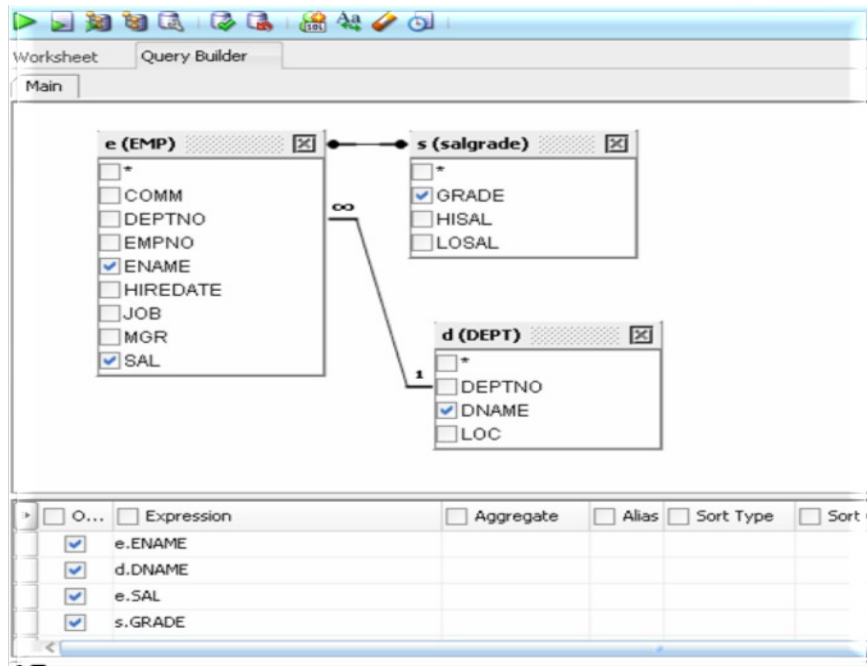
```
SELECT
  e.ename,
  d.dname,
```

```

    e.sal,
    s.grade
FROM
    emp e
INNER JOIN salgrade s
ON
    e.sal BETWEEN s.losal AND s.hisal
INNER JOIN dept d
ON
    e.deptno = d.deptno

```

Построению такого и подобных запросов с многократными соединениями помогает графический инструментарий Query Builder, встроенный в SQL Developer в версиях 3.1+:



Тем не менее, применение в одном запросе многократных соединений омрачается ухудшением читаемости и падением скорости обработки. Читаемость запроса можно пытаться «спасти» с помощью аппарата view и разложением на подзапросы (и то и другое рассматривается ниже), а скорость обработки в Oracle обычно заметно падает при числе соединений пять и более.

Наконец, наличие соединения во фразе FROM не препятствует указанию других источников данных обычным образом. Следующий запрос не имеет практического смысла и приводится только для показа самой возможности:

```

SELECT *
FROM   dept
       NATURAL INNER JOIN
       dept1
       , emp
WHERE  loc <> 'NEW YORK'
;

```

Заметьте, однако, что здесь возникло декартово произведение, так что трудность приведения содержательного примера возникла не случайно. С формальной же точки зрения вместо перечисления через запятую тут правильнее применить CROSS JOIN.

Синтаксис допускает определенные вольности. Так, ключевые слова INNER и OUTER необязательны и не влияют на смысл операции соединения в SQL. Круглые скобки для логического условия после ON необязательны.

6.12.4.5. Замечание по поводу использования Query Builder

Это средство SQL Developer облегчает написание запросов, однако у легкости есть оборотная сторона. Она может подталкивать программиста к быстрым решениям, правильным содержательно, но не обязательно наиболее эффективным. В качестве упражнения предлагается построить с его помощью несколько измененный запрос, приводившийся выше, с самосоединением EMP. Помимо сведений, кто кому подчиняется, предлагается выдать рядом с именами сотрудников названия их отделов (взятых из DEPT).

Упражнение. Составить запрос, возвращающий сведения о подчиненности сотрудников примерно в таком виде:

ENAME	DNAME	'->'	ENAME	DNAME
-----	-----	-----	-----	-----
CLARK	ACCOUNTING	->	MILLER	ACCOUNTING
KING	ACCOUNTING	->	CLARK	ACCOUNTING
KING	ACCOUNTING	->	JONES	RESEARCH
FORD	RESEARCH	->	SMITH	RESEARCH
...

Попробовать оценить формулировку запроса. Можно ли ее улучшить?

6.12.5. Сравнение собственного синтаксиса Oracle SQL для открытых соединений и стандартного

Синтаксис '(+)' сосуществует в Oracle SQL со стандартным OUTER JOIN для полуоткрытых соединений, а закрытые, полностью открытые соединения и декартовы произведения могут формулироваться вовсе без INNER, FULL и CROSS JOIN. Возникает неопределенность: стоит ли программисту вообще полагаться на синтаксис SQL:1999, а в случае открытых соединений использовать '(+)'?

Фирма Oracle, вслед за комитетами по стандартизации SQL, рекомендует использовать синтаксис SQL:1999. В общем это оправдано, и можно дополнительно посоветовать использовать ON вместо USING там, где это возможно, и избегать NATURAL JOIN.

Стандартный синтаксис для соединений — вероятно, одно из самых удачных решений в SQL, особенно если учесть дополнительно сферу его употребления: выражаясь фигурально, в 99% случаев запрос более чем к одной таблице в SQL будет ни чем иным, как соединением, и в оставшемся условном 1% случаев — декартовым произведением. Многословность синтаксиса SQL:1999 преднамеренная, она обращает внимание программиста на наличие в запросе соединения и снижает, тем самым, вероятность ошибок.

Что касается сравнения конструкций '(+)' и OUTER JOIN, то по факту они не равноценны. Синтаксис с '(+)' допускает некоторые формулировки, которые не поддерживаются в OUTER JOIN, и такие случаи требуют внимания.

6.12.5.1. Соединение таблицы с развернутым ее столбцом типа «коллекция»

Обозначение '(+)' не допускает переформулировку с использованием OUTER JOIN, то есть незаменимо, при выполнении довольно своеобразного соединения таблицы с табличными данными, полученными разворачиванием данных-коллекций, извлеченных из таблицы:

```
SELECT ... FROM таблица, TABLE ( столбец-коллекция_из_таблицы ) (+) WHERE ....
```

Общее представление о коллекциях дается в материале об объектных возможностях в Oracle.

6.12.5.2. «Полусоединение с константой»

Синтаксис запроса с '(+)', в отличие от OUTER JOIN, поддерживает «полусоединение с константой» (с постоянным значением, в общем случае с результатом постоянного выражения). Пример:

```
SQL> SELECT * FROM dept WHERE deptno ( + ) = 50;
```

no rows selected

(Заодно это пример того, как в постоянном выражении *не* допускается скалярный подзапрос, даже к «постоянной» таблице DUAL, не говоря уже о подзапросе к обыкновенной таблице из БД. В этом легко удостовериться, попытавшись подставить такой подзапрос вместо постоянной величины 50.)

Обратите внимание, что «настоящее» полусоединение ведет себя иначе:

```
SQL> SELECT *
  2  FROM dept, ( SELECT 50 c FROM dual ) t
  3  WHERE deptno ( + ) = t.c;

DEPTNO DNAME LOC C
-----
50
```

(Переформулировка последнего запроса под LEFT OUTER JOIN, как это и должно быть, даст тот же ответ.)

Смысл однотабличного полуоткрытого соединения с постоянной сомнителен, однако пример показывает, нарушения синтаксиса в нем Oracle не видит, и это затрудняет выявление ошибок (содержательных, а не синтаксических) программирования. Подобное имеет место с декартовым произведением.

В то же время, когда речь идет о цепочке соединений, полусоединение с постоянной уже обретает смысл, в то время как проблема содержательно неправильной формулировки остается:

SELECT ... FROM t1, t2, t3 WHERE t1.c1 = t2.c1 (+) AND t2.c2 = t3.c2 (+) AND t3.c3 = 'Y'	<i>Должно быть</i> →	SELECT ... FROM FROM t1, t2, t3 WHERE t1.c1 = t2.c1 (+) AND t2.c2 = t3.c2 (+) AND t3.c3 (+) = 'Y'
--	-------------------------	---

Пример. Запрос:

```
SELECT d.dname, e.ename, e.job
FROM dept d, emp e
WHERE d.deptno = e.deptno ( + )
AND e.job = 'SALESMAN'
;
```

Ответ:

DNAME	ENAME	JOB
SALES	ALLEN	SALESMAN
SALES	TURNER	SALESMAN
SALES	MARTIN	SALESMAN
SALES	WARD	SALESMAN

Запрос:

```
SELECT d.dname, e.ename, e.job
FROM dept d, emp e
WHERE d.deptno = e.deptno ( + )
AND e.job ( + ) = 'SALESMAN'
```

;

Ответ:

DNAME	ENAME	JOB

ACCOUNTING		
RESEARCH		
SALES	ALLEN	SALESMAN
SALES	TURNER	SALESMAN
SALES	MARTIN	SALESMAN
SALES	WARD	SALESMAN
OPERATIONS		

Отсутствие '(+)' в условном выражении сравнения с постоянной величиной фактически «закрывает» полуоткрытое (по мысли программиста) соединение, делая ненужным простановку '(+)' в других местах выражения после WHERE.

Подобное «закрытие» полуоткрытого соединения по результату не должно отличаться от «честного» закрытого соединения, в котором '(+)' отсутствует вовсе, однако «половинчатая» формулировка при этом может вызывать более затратные планы обработки, нежели «честно» закрытая.

Упражнение. Выполнить последние два запроса, заменив название существующей должности SALESMAN на несуществующей, к примеру, CEO. Объяснить смысл обоих при том ответов системы.

6.12.5.3. Многостолбцовые открытые соединения

В-третьих, синтаксис '(+)' допускает опять-таки содержательно неправомерные формулировки в многостолбцовых открытых соединениях:

SELECT ... FROM t1, t2 WHERE t1.c1 = t2.c1 (+) AND t1.c2 = t2.c2	Должно быть →	SELECT ... FROM t1, t2 WHERE t1.c1 = t2.c1 (+) AND t1.c2 = t2.c2 (+) или SELECT ... FROM t1 LEFT OUTER JOIN t2 USING (c1, c2) или SELECT ... FROM t1 LEFT OUTER JOIN t2 ON t1.c1, t2.c2
---	------------------	--

Пример:

```
DROP TABLE e PURGE;
DROP TABLE d PURGE;
CREATE TABLE e
  ( ename VARCHAR2 ( 10 ), loc VARCHAR2 ( 13 ), deptno NUMBER ( 2 ) )
;
CREATE TABLE d
  ( dname VARCHAR2 ( 14 ), loc VARCHAR2 ( 13 ), deptno NUMBER ( 2 ) )
;
INSERT INTO e VALUES ( 'SMITH', 'DALLAS', 20 );
INSERT INTO e VALUES ( 'SCOTT', 'DALLAS', 30 );
INSERT INTO e VALUES ( 'ALLEN', 'BOSTON', 20 );
INSERT INTO e VALUES ( 'MILLER', NULL, NULL );

INSERT INTO d VALUES ( 'SALES', 'DALLAS', 20 );
INSERT INTO d VALUES ( 'RESEARCH', 'DALLAS', 40 );
INSERT INTO d VALUES ( 'OPERATIONS', 'CHICAGO', 20 );
```

```
INSERT INTO d VALUES ( 'ACCOUNTING', NULL, NULL );
```

Здесь, в отличие от DEPT, отделы определяются парой <имя, номер>. Имеются сотрудники, работающие в несуществующих отделах, сотрудники, вовсе не работающие, отделы, в которых не числятся сотрудников и отделы, для которых отождествитель <имя, номер> не задан.

Проверка неправильной и правильной формулировок на запросе сведений о сотрудниках, работающих в имеющихся отделах:

```
SQL> SELECT e.ename, d.dname, e.loc, d.loc, e.deptno, d.deptno
2 FROM d, e
3 WHERE d.deptno = e.deptno ( + ) AND d.loc = e.loc;
```

ENAME	DNAME	LOC	LOC	DEPTNO	DEPTNO
SMITH	SALES	DALLAS	DALLAS	20	20

```
SQL> SELECT e.ename, d.dname, e.loc, d.loc, e.deptno, d.deptno
2 FROM d, e
3 WHERE d.deptno = e.deptno ( + ) AND d.loc = e.loc ( + );
```

ENAME	DNAME	LOC	LOC	DEPTNO	DEPTNO
SMITH	SALES	DALLAS	DALLAS	20	20
	ACCOUNTING				
	RESEARCH		DALLAS		40
	OPERATIONS		CHICAGO		20

Как видно, отсутствие повторения '(+)' не нарушает синтаксиса, однако фактически «закрывает» соединение.

Замечательно, что формулировки со стандартным OUTER JOIN позволяют задать только правильный запрос.

```
SELECT e.ename, d.dname, loc, deptno
FROM d LEFT OUTER JOIN e USING ( deptno, loc )
;
```

Ответ:

ENAME	DNAME	LOC	DEPTNO
SMITH	SALES	DALLAS	20
	ACCOUNTING		
	RESEARCH	DALLAS	40
	OPERATIONS	CHICAGO	20

```
SELECT e.ename, d.dname, e.loc, d.loc, e.deptno, d.deptno
FROM d LEFT OUTER JOIN e ON d.deptno = e.deptno
AND d.loc = e.loc
;
```

Ответ:

ENAME	DNAME	LOC	LOC	DEPTNO	DEPTNO
SMITH	SALES	DALLAS	DALLAS	20	20
	ACCOUNTING				
	OPERATIONS		CHICAGO		20
	RESEARCH		DALLAS		40

Запросы с OUTER JOIN ... ON и с повторением '(+)' позволяют понять, какие строки результата обязаны отделам, в которых нет сотрудников.

Понять логику вычисления ответа помогает декартово произведение таблиц D и E:

```
SQL> SELECT e.ename, e.loc, e.deptno, d.dname, d.loc, d.deptno
2 FROM d CROSS JOIN e;
```

ENAME	LOC	DEPTNO	DNAME	LOC	DEPTNO
SMITH	DALLAS	20	SALES	DALLAS	20
SCOTT	DALLAS	30	SALES	DALLAS	20
ALLEN	BOSTON	20	SALES	DALLAS	20
MILLER			SALES	DALLAS	20
SMITH	DALLAS	20	RESEARCH	DALLAS	40
SCOTT	DALLAS	30	RESEARCH	DALLAS	40
ALLEN	BOSTON	20	RESEARCH	DALLAS	40
MILLER			RESEARCH	DALLAS	40
SMITH	DALLAS	20	OPERATIONS	CHICAGO	20
SCOTT	DALLAS	30	OPERATIONS	CHICAGO	20
ALLEN	BOSTON	20	OPERATIONS	CHICAGO	20
MILLER			OPERATIONS	CHICAGO	20
SMITH	DALLAS	20	ACCOUNTING		
SCOTT	DALLAS	30	ACCOUNTING		
ALLEN	BOSTON	20	ACCOUNTING		
MILLER			ACCOUNTING		

Пары <DALLAS, 40>, <CHICAGO, 20> и <NULL, NULL>, задающие три отдела, отсутствуют в соединяемых столбцах сотрудников, поэтому (соединение полуоткрыто) они добавили к декартову произведению три искусственные строки:

NULL	NULL	NULL	RESEARCH	DALLAS	40
NULL	NULL	NULL	OPERATIONS	CHICAGO	20
NULL	NULL	NULL	ACCOUNTING		

Полученное таким образом множество строк поступило на обработку фразе WHERE.

С другой стороны, содержательная неправомерность записи многостолбцового полуоткрытого соединения с хотя бы одним «забытым» указанием '(+)' прояснится, если вспомнить о другой, содержательно тождественной правильной формулировке:

```
SELECT e.ename, e.loc, e.deptno, d.dname, d.loc, d.deptno
FROM d, e
WHERE d.deptno = e.deptno AND d.loc = e.loc
UNION ALL
SELECT NULL, NULL, NULL, d.dname, d.loc, d.deptno
FROM d
WHERE NOT EXISTS
( SELECT e.ename
FROM e
WHERE e.deptno = d.deptno AND e.loc = d.loc )
;
```

6.13. Подзапросы и разложение запроса на подзапросы

6.13.1. Подзапросы в тексте запроса

Обычные, вложенные, подзапросы (запросы внутри запросов) могут быть:

- скалярными, возвращающими одно значение какого-то типа (не обязательно простого, а например составного, объектного): формально — одностолбцовыми и одно- либо нульстрочными;
- однострочными, возвращающими набор значений в форме строки;
- многострочными, возвращающими произвольное множество строк.

Подзапросы этих категорий могут возникать в разных местах предложения SELECT и предложений DML по изменению данных (рассматриваются далее):

- в выражениях в качестве значения (однозначные)
- в условных выражениях (WHERE или CASE) как операнд сравнения (однозначные)
- в условных выражениях (WHERE или CASE) как операнд сравнения со списком (многостолбцовые однострочные)
- в условных выражениях (WHERE или CASE) как операнд сравнения в операторах сравнения с кванторами ANY и ALL, IN с подзапросом (многострочные)
- во фразе SET предложения UPDATE (однозначные и многостолбцовые однострочные)
- в предложении INSERT INTO ... AS SELECT (многострочные)
- в предложениях SELECT, INSERT, UPDATE, DELETE, MERGE везде, где разрешено указывать имена таблиц (многострочные)

Зоны видимости имен таблиц и их столбцов при использовании вложенных подзапросов поясняется следующим примером:

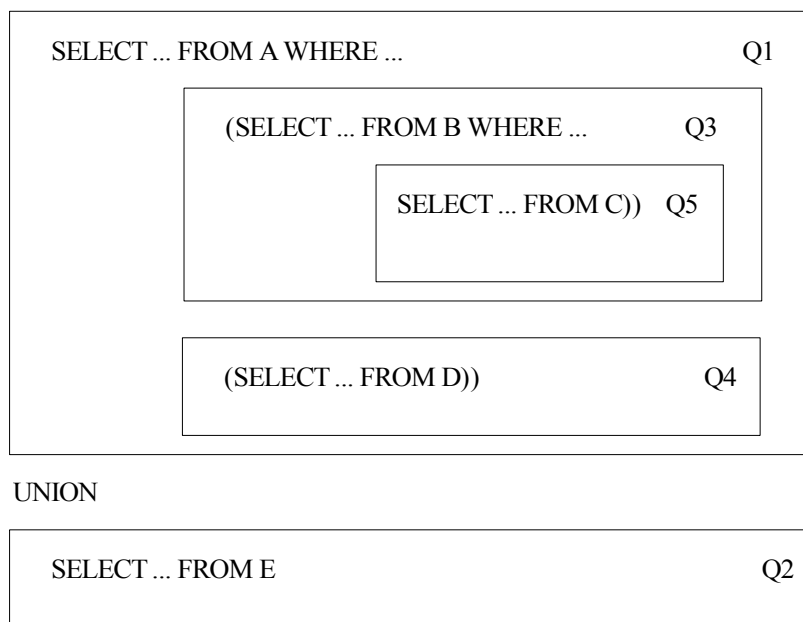


Таблица A видна из Q1, Q3, Q4, Q5. Таблица B видна из Q3, Q5.

Указание ORDER BY в подзапросе имеет смысл только в *одном*, особом случае «запроса с квотой»: типа *TopN* (отбор первых N записей). Любопытно, что именно в этом случае сортировка технически в полном объеме выполняться как раз не будет, в то время как в остальных применениях ORDER BY в подзапросе, несмотря на бессмысленность, будет!

6.13.2. Вынесенные подзапросы, или разложение запроса на подзапросы с помощью фразы WITH

Oracle допускает вынесение определений подзапросов из тела основного запроса с помощью особой фразы WITH. Эта техника получила название subquery factoring, то есть «факторизация», «разложение на подзапросы».

Фраза WITH используется в двух целях:

- для придания запросу формулировки, более понятной программисту (просто subquery factoring), и
- для записи рекурсивных запросов (recursive subquery factoring).

Обе формулировки фразы WITH не противоречат друг другу и могут использоваться совместно. Первый вариант фразы WITH не отменяет описательного характера предложения SELECT и (помимо удобства формулировки) способен разве что дать ускоренное общее выполнение. Рекурсивный же вариант фразы WITH по сути откровенно процедурен и тем противоречит описательному характеру предложения SELECT, положенному когда-то в основу SQL.

Простое и рекурсивное разложение на подзапросы с помощью фразы WITH рассматриваются ниже.

6.13.3. Вынесение определений подзапросов ради удобства формулировки

Возможность была введена в версии 9.0 в соответствии со стандартом SQL:1999. В стандарт же она попала из правил построения выражений над отношениями в реляционной теории. Фраза WITH в этом качестве — неисполняемая и предназначена в первую очередь для придания тексту сложного запроса более понятную структуру. Но сверх этого она *может* способствовать более эффективному вычислению СУБД ответа на запрос, сообщая о нем дополнительную информацию.

Фраза WITH предшествует фразе SELECT и позволяет привести сразу несколько предваряющих формулировок подзапросов для ссылки на них в нижеформулируемом основном запросе. Идея употребления демонстрируется следующей схемой:

```
WITH
  x AS ( SELECT ... )
, y AS ( SELECT ... FROM x )
, z AS ( SELECT ... FROM x, y )
SELECT ... FROM x, y, z, w
;
```

Пример употребления:

```
WITH
  commissioners AS ( SELECT * FROM emp WHERE comm IS NOT NULL )
SELECT
  ename
, deptno
, sal + comm AS earnings
FROM commissioners
;
```

Еще пример, с переформулировкой одного из приводившихся ранее запросов с «разворачиванием» данных одного столбца в несколько:

```
WITH baseview AS ( SELECT job, deptno FROM emp )
SELECT *
FROM baseview
PIVOT ( COUNT ( * ) FOR deptno IN ( 10, 30 ) )
;
```

Следующий пример позволяет достаточно удобно выдать отчет о найме сотрудников на работу в нужный период времени, не прибегая к созданию опорной таблицы в БД, как это делалось в одном из примеров выше:

```
WITH pivot_years AS (
  SELECT ( ROWNUM - 1 ) + 1980 AS year
  FROM dual
  CONNECT BY ROWNUM <= 10
)
SELECT p.year, COUNT ( e.empno )
FROM pivot_years p LEFT OUTER JOIN emp e
ON ( p.year = EXTRACT ( YEAR FROM e.hiredate ) )
GROUP BY p.year
```

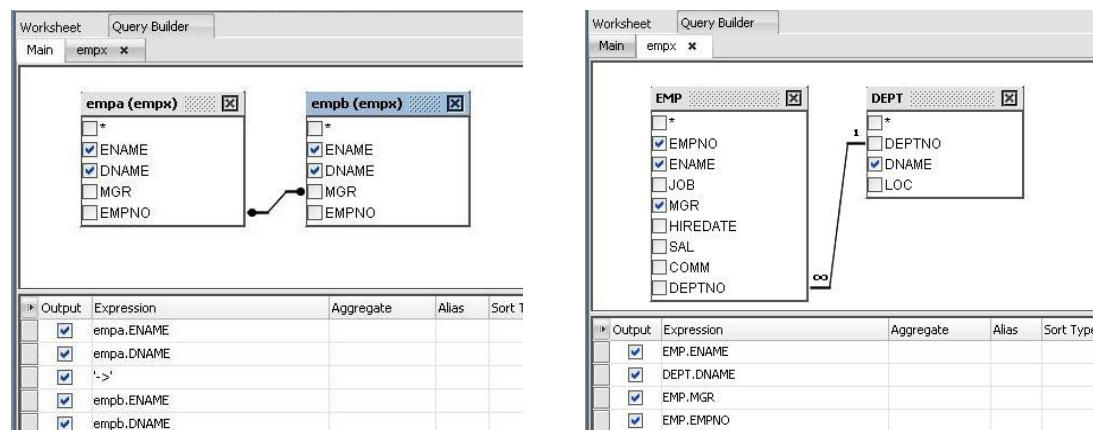
```
ORDER BY p.year
;
```

Следующий пример позволяет пользователю SYS выдать сведения о десяти запросах к БД, более всех остальных выполняющих логические обращения к диску:

```
WITH buffergets AS (
SELECT
    u.username
  , q.buffer_gets
  , q.executions
  , q.buffer_gets / CASE q.executions WHEN 0 THEN 1
                      ELSE q.executions
                      END
    "read/exec ratio"
  , q.command_type
  , q.sql_text
FROM
    v$sqlarea q
  , dba_users u
WHERE
    q.parsing_user_id = u.user_id
ORDER BY 2 DESC
)
SELECT * FROM buffergets WHERE ROWNUM <= 10
;
```

В процессе обработки подзапрос с предваряющей формулировкой вычисляется как неименованное представление данных («вписанное в запрос представление», inline view), то есть либо самостоятельно, либо опосредованно, в результате «растворения» в теле основного запроса после общего внутреннего переформулирования.

Фразу WITH программисту помогает использовать Query Builder в SQL Developer, например:



Это пример построения запроса, возвращающего примерно такой результат:

ENAME	DNAME	'->'	ENAME	DNAME
CLARK	ACCOUNTING	->	MILLER	ACCOUNTING
KING	ACCOUNTING	->	CLARK	ACCOUNTING
KING	ACCOUNTING	->	JONES	RESEARCH
FORD	RESEARCH	->	SMITH	RESEARCH
...

Однако применение в так составленном самосоединении фразы WITH не только улучшает восприятие текста человеком, но и дает возможность оптимизатору запросов улучшить план выполнения.

6.13.4. Формулирование рекурсивных запросов

С версии 11.2 фраза WITH может использоваться для формулирования рекурсивных запросов, в соответствии (неполном) со стандартом SQL:1999. В этом качестве она способна решать ту же задачу, что и CONNECT BY, однако (а) делает это по-другому с СУБД других типов образом, (б) обладает более широкими возможностями, (в) применима не только к запросам по иерархии и (г) записывается значительно более замысловато.

Общий алгоритм вычисления фразой WITH таков:

```
Результат := пусто;
Добавок := якорный SELECT ...;
пока Добавок не пуст выполнять:
    Результат := Результат UNION ALL Добавок;
    Добавок := рекурсивный SELECT ... FROM Добавок ...;
конец цикла;
```

Предложение SELECT для определения исходного множества строк называется якорным (anchor — зацепным, опорным) членом фразы WITH. Предложение SELECT для получения добавочного множества строк Oracle называет рекурсивным членом.

6.13.4.1. Простой пример

Простой пример употребления фразы WITH для построения рекурсивного запроса:

```
WITH
numbers ( n ) AS (
    SELECT 1 AS n FROM dual -- исходное множество -- одна строка
    UNION ALL
    SELECT n + 1 AS n      -- символическое «объединение» строк
    FROM   numbers        -- рекурсия: добавок к предыдущему результату
    WHERE  n < 5           -- предыдущий результат в качестве источника данных
    -- если не ограничить, будет бесконечная рекурсия
)
SELECT n FROM numbers      -- основной запрос
;
```

Операция UNION ALL здесь используется символически, в рамках определенного контекста, для указания способа рекурсивного накопления результата.

Ответ:

```
----- N
1
2
3
4
5
```

Строка с N = 1 получена из якорного запроса, а остальные строки — из рекурсивного. Из примера видна обратная сторона рекурсивных формулировок: при неаккуратном планировании они допускают «бесконечное» выполнение (на деле — пока хватит ресурсов СУБД для сеанса или же пока администратор не прервет запрос или сеанс). Программист обязан отнестись к построению рекурсивного запроса ответственно.

Еще один вывод из этого примера: подобно случаю с CONNECT BY, вынесенный рекурсивный подзапрос применим вовсе не обязательно только к иерархически организованным данным. В приведенном запросе обращения к данным по сути вообще нет, хотя источник формально и указан.

Пример с дополнительным разъяснением способа выполнения:

```
SQL> WITH
  2     anchor1234 ( n ) AS (                -- обычный
  3         SELECT 1 FROM dual UNION ALL
  4         SELECT 2 FROM dual UNION ALL
  5         SELECT 3 FROM dual UNION ALL
  6         SELECT 4 FROM dual
  7     )
  8 , numbers ( n ) AS (                    -- рекурсивный
  9     SELECT n FROM anchor1234
 10     UNION ALL
 11     SELECT n + 1 AS n
 12     FROM   numbers
 13     WHERE  n < 5
 14 )
15 SELECT n FROM numbers
16 ;
```

```

----- N
1  ← якорный запрос
2  ← якорный запрос
3  ← якорный запрос
4  ← якорный запрос
2  ← рекурсия 1
3  ← рекурсия 1
4  ← рекурсия 1
5  ← рекурсия 1
3  ← рекурсия 2
4  ← рекурсия 2
5  ← рекурсия 2
4  ← рекурсия 3
5  ← рекурсия 3
5  ← рекурсия 4
```

Приведенный пример рекурсивного запроса позволяет перестроить один из приводившихся ранее «отчетных» запросов без прибегания к служебной таблице (названной выше PIVOT_YEARS) или же к табличной функции:

```
WITH     period ( year ) AS (
        SELECT 1980 AS year FROM dual
        UNION ALL
        SELECT year + 1 AS year
        FROM   period
        WHERE  year < 1990
    )
SELECT  p.year, COUNT ( e.empno )
FROM    emp e RIGHT OUTER JOIN period p
        ON p.year = EXTRACT ( YEAR FROM e.hiredate )
GROUP BY p.year
ORDER BY p.year
;
```

Запрос в приведенной формулировке самодостаточен. Желание его параметризовать, буде оно появится, осуществимо с помощью «контекста сеанса» Oracle, но уже потребует соблюдения определенной технологии программирования при обращении к запросу.

6.13.4.2. Использование предыдущих значений при рекурсивном вычислении

Рекурсивные запросы с фразой WITH позволяют программисту больше, нежели запросы с CONNECT BY (тоже рекурсивные). Например, они позволяют накапливать изменения и не испытывают необходимости в функциях LEVEL или SYS_CONNECT_BY_PATH, имея возможность несложно их моделировать.

Пример запроса по маршрутам из Москвы с подсчетом километража:

```
WITH stepbystep ( node, route, distance ) AS (
  SELECT node, parent || '-' || node, distance
  FROM   ways
  WHERE  parent = 'Москва'
  UNION ALL
  SELECT w.node
        , s.route || '-' || w.node
        , w.distance + s.distance
  FROM   ways w
        INNER JOIN
        stepbystep s
        ON ( s.node = w.parent )
)
SELECT route, distance FROM stepbystep
/
```

Ответ:

ROUTE	DISTANCE
Москва-Ленинград	696
Москва-Новгород	538
Москва-Новгород-Ленинград	717
Москва-Ленинград-Выборг	831
Москва-Новгород-Ленинград-Выборг	852

Запрос по маршрутам из Выборга аналогичен, но с поправкой на симметрию, вызванную движением по иерархии снизу вверх, а не сверху вниз:

```
WITH stepbystep ( parent, route, distance ) AS (
  SELECT parent, node || '-' || parent, distance
  FROM   ways
  WHERE  node = 'Выборг'
  UNION ALL
  SELECT w.parent
        , s.route || '-' || w.parent
        , w.distance + s.distance
  FROM   ways w
        INNER JOIN
        stepbystep s
        ON ( s.parent = w.node )
)
SELECT route, distance FROM stepbystep
/
```

Ответ:

ROUTE	DISTANCE
Выборг-Ленинград	135
Выборг-Ленинград-Москва	831
Выборг-Ленинград-Новгород	314
Выборг-Ленинград-Новгород-Москва	852

6.13.4.3. Обработка заикленности данных

Пример организации заикленности в сведениях о маршрутах:

```
INSERT INTO ways VALUES ( 'Новгород', 'Выборг', 135 );
```

Реакция на появление цикла (уже получается не иерархия) в этом случае отлична от имевшейся для CONNECT BY и будет:

```
ERROR:
ORA-32044: cycle detected while executing recursive WITH query
```

Упражнение. Проверить это самостоятельно.

Для предупреждения заикливания вычислений вводится специальное указание CYCLE, где следует указать перечень (в общем случае) столбцов для распознавания хождения по кругу, придумать название столбца-индикатора (он автоматически включается в конечный ответ) и задать пару символов: для обозначения незаикленной строки и для обозначения строки, где было зафиксировано повторение значений в различительных столбцах:

```
WITH stepbystep ( node, route, distance ) AS (
  SELECT node, parent || '-' || node, distance
  FROM   ways
  WHERE  parent = 'Москва'
  UNION ALL
  SELECT w.node
        , s.route || '-' || w.node
        , w.distance + s.distance
  FROM   ways w
        INNER JOIN
        stepbystep s
        ON ( s.node = w.parent )
)
CYCLE node SET cyclemark TO 'X' DEFAULT '-'
SELECT route, distance, cyclemark FROM stepbystep
/
```

Ответ:

ROUTE	DISTANCE	C
Москва-Ленинград	696	-
Москва-Новгород	538	-
Москва-Новгород-Ленинград	717	-
Москва-Ленинград-Выборг	831	-
Москва-Новгород-Ленинград-Выборг	852	-
Москва-Ленинград-Выборг-Новгород	966	-
Москва-Ленинград-Выборг-Новгород-Ленинград	1145	X
Москва-Новгород-Ленинград-Выборг-Новгород	987	X

6.13.4.4. Упорядочение результата

Для придания порядка строкам результата в запросах с CONNECT BY используется собственная конструкция ORDER BY SIBLINGS. Аналогичным образом в вынесенном рекурсивном запросе применяется особое указание SEARCH. В его рамках программистом задается в том числе вымышленное имя столбца, в котором СУБД автоматически проставит числовые значения, и который самостоятельно включит в порождаемый набор столбцов. На этот столбец программист может сослаться далее уже в обычной фразе ORDER BY для создания нужного порядка строк. Пример:

```

ROLLBACK;

WITH stepbystep ( node, route, distance ) AS (
  SELECT node, parent || '-' || node, distance
  FROM   ways
  WHERE  parent = 'Москва'
  UNION ALL
  SELECT w.node
        , s.route || '-' || w.node
        , w.distance + s.distance
  FROM   ways w
        INNER JOIN
        stepbystep s
        ON ( s.node = w.parent )
)
  SEARCH DEPTH FIRST BY node DESC SET orderval
SELECT  route, distance, orderval
FROM    stepbystep
ORDER BY orderval DESC
/

```

Ответ:

ROUTE	DISTANCE	ORDERVAL
Москва-Ленинград-Выборг	831	5
Москва-Ленинград	696	4
Москва-Новгород-Ленинград-Выборг	852	3
Москва-Новгород-Ленинград	717	2
Москва-Новгород	538	1

Подробности и прочие свойства построений указания SEARCH приведены в документации по Oracle.

6.13.4.5. Замечание об общей формулировке запроса

Общая формулировка рекурсивного запроса в стандарте SQL и в Oracle способна вызвать у некоторых программистов недоумение, однако она имеет свое вероятное обоснование. Ранее упоминалось о возможности описания реляционной БД средствами логики предикатов. В таком случае база представляет из себя набор истинных утверждений. Пусть есть «предикатный символ» (predicate symbol, то есть «обозначение утверждения») *way* (*x*, *y*) как общее обозначение однотипных утверждений (километраж и вероятные другие свойства здесь для простоты опущены как несущественные). В БД представлено несколько конкретных соответствующих ему истинных утверждений, например:

```

ways ( 'Ленинград', 'Выборг' )
ways ( 'Новгород', 'Ленинград' )
...

```

То есть «имеется путь от Ленинграда до Выборга», «от Новгорода до Ленинграда» и так далее. Это так называемое «существовательное» (intensional) определение БД, явно перечисляющее объекты с их свойствами. Дополнительно можно ввести еще один предикатный символ *route* (*x*, *y*) со смыслом «маршрут». Допустим, что утверждения для него представлены не «существовательно», а «расширительно» (extensionally), в виде двух правил вывода:

```

route ( x, y ) → ways ( x, y )
route ( x, y ) → route ( x, z ), ways ( z, y )

```

Это позволяет получать из БД сведения (о «маршрутах»), напрямую в ней не представленные, и БД становится «расширительной». Так, она оказалась дополнена группой новых утверждений вида *route* (*x*, *y*).

Теперь если обозначить *ways* (*x*, *y*) как *W*, *route* (*x*, *y*) как *R*, второе (рекурсивное) правило вывода как \bullet , то с позиций уже реляционной алгебры для новых сведений из БД для определения маршрутов можно предложить формулировку $R = W \cup R \bullet W^5$. Она удивительно напоминает общее построение рекурсивного запроса в SQL, где однако пошли дальше и обобщили операцию \cup объединения множеств на упомянутую группу. Если обобщенную множественную операцию указать как \circ , формула будет выглядеть как $R = W \circ R \bullet W$.

Получается, что рекурсивная формулировка запроса в SQL придает вообще-то «существительной» базе, предполагаемой этим языком, некоторые качества «расширительной», где возможно получение новых «знаний» из имеющихся. К сожалению, на практике такое достижение нельзя подкрепить созданием представления данных (*view*) на основе рекурсивного запроса ввиду имеющегося в настоящее время в Oracle запрета на подобное действие.

Оборотной стороной, помимо риска заикленности (воизбежание которого Oracle дает упоминавшееся частичное решение), является пониженная производительность вычисления. Для повышения производительности Oracle тоже предлагает определенную гамму решений (организация *materialized view* и прочее), но все они носят неполный характер и обременены собственными издержками.

⁵ Идея получения подобной формулы упоминается в книге Марков А. С., Лисовский К. Ю. Базы данных: Введение в теорию и методологию. // Москва: Финансы и Статистика, 2006, интересной разработчику и программисту БД во многих других отношениях.

7. Обновление данных в таблицах

Операции SQL по изменению данных в БД — это принадлежащие категории DML INSERT, UPDATE и DELETE плюс вторичная по отношению к ним MERGE. Все они множественные (вслед за реляционной моделью, где это неслучайно), то есть в общем рассчитаны одним действием изменить сразу множество строк таблицы. Исключение составляет формально однострочная разновидность оператора INSERT.

Операции INSERT, UPDATE, DELETE и MERGE унаследованы в Oracle от стандарта SQL. В реляционной теории операций INSERT, UPDATE и DELETE как таковых нет, но они легко моделируются существующими другими.

7.1. Добавление новых строк

7.1.1. Добавление одной строки

Пример:

```
INSERT INTO proj ( projno ) VALUES ( 20 );
```

В общем случае имена столбцов (до слова VALUES) и значения (после слова VALUES) приводятся списками с равными количествами элементов.

Имена столбцов со свойством NOT NULL и значения для них должны присутствовать в списках обязательно, если только для них не определены умолчательные значения или если значения не заносятся триггерными процедурами. При несоблюдении этих условий возникает ошибка времени исполнения.

В то время как порядок перечисления выражений должен копировать порядок перечисления имен столбцов перед словом VALUES, сам порядок имен столбцов может быть произвольным. Об этом напоминает синтаксис, и это еще одно наследие в SQL реляционной теории. Следующие предложения равносильны:

```
INSERT  
INTO proj ( projno, pname, bdate, budget )  
VALUES  
    ( 30, 'BETA', SYSDATE, 20000 )  
;
```

```
INSERT  
INTO proj ( bdate, budget, projno, pname )  
VALUES  
    ( SYSDATE, 20000, 30, 'BETA' )  
;
```

Для последней пары запросов SQL допускает и более краткую формулировку:

```
INSERT INTO proj VALUES ( 30, 'BETA', SYSDATE, 20000 );
```

Эта краткость может показаться соблазнительной, однако подобная формулировка недопустима в реляционной БД, так как предполагает наличие порядка столбцов в таблице SQL, в то время как в отношениях реляционной модели порядка атрибутов не существует. С практической точки зрения полагаться в запросе на порядок столбцов — ненадежно, ведь Oracle разрешает добавлять и удалять столбцы в таблице; так что со временем порядок может нарушиться и операция окажется некорректной. Кроме того, перечисляя выражения для подстановки значений в поля добавляемой строки, программист, не имея перед глазами списка имен полей, легко может ошибиться.

Пример использования скалярного подзапроса в выражении во фразе VALUES:

```

INSERT INTO emp
  ( empno, deptno )
VALUES
  ( 1111, ( SELECT deptno FROM dept WHERE loc = 'CHICAGO' ) )
;

```

На деле пример относится лишь к использованию скалярного подзапроса в построении выражения.

7.1.2. Добавление строк, полученных подзапросом

Множественный вариант INSERT предполагает добавление одним оператором в таблицу сразу группы строк. Добавляемые строки определяются оператором SELECT. Пример:

```

DELETE FROM dept_copy;

INSERT INTO dept_copy SELECT * FROM dept WHERE deptno IN ( 10, 20 );

INSERT
  INTO dept_copy ( loc, dname, deptno )
  SELECT loc, INITCAP ( dname ), deptno + 100 FROM dept
;

INSERT INTO dept_copy ( deptno ) SELECT deptno + 200 FROM dept
;

```

Замечания

- На формулировку SELECT в предложении INSERT никаких нарочных ограничений не накладывается (в том числе допускаются GROUP BY, агрегатные функции и прочее)
- Типы столбцов должны быть совместимы.

Операция вставки INSERT в общем случае множественная. Ее однострочный вариант можно полагать частным случаем множественного.

7.1.3. Добавление строк одним оператором в несколько таблиц

С версии 9 строки, полученные подзапросом, можно «раскидать» по нескольким таблицам единственным оператором INSERT, например:

```

CREATE TABLE e1000 AS SELECT ename, sal FROM emp WHERE 1 = 2;

CREATE TABLE e2000 AS SELECT ename, sal FROM emp WHERE 1 = 2;

CREATE TABLE e3000 AS SELECT * FROM emp WHERE 1 = 2;

INSERT ALL
  WHEN sal > 3000 THEN INTO e3000
  WHEN sal > 2000 THEN INTO e2000 ( ename ) VALUES ( ename )
  WHEN sal > 1000 THEN INTO e1000 VALUES ( ename, sal )
SELECT * FROM emp WHERE job <> 'SALESMAN'
;

```

Упражнение. Проверить содержимое таблиц E1000, E2000 и E3000. Сохранить их данные, удалить таблицы и выполнить пример заново, указав вместо INSERT ALL слова INSERT FIRST. Сравнить новое содержимое таблиц с предыдущим.

Последовательность проверок WHEN можно завершать проверкой ELSE.

Фразы WHEN ... THEN можно опускать, тогда будет выполняться безусловная вставка строк в таблицы, например:

```
INSERT ALL
  WHEN comm IS NULL THEN INTO e1000 VALUES ( ename, sal )
  INTO e2000 ( ename ) VALUES ( ename )
  INTO e3000
SELECT * FROM emp WHERE job <> 'SALESMAN'
;
```

Вставка строк оператором INSERT разом в несколько таблиц эффективнее последовательности однотабличных INSERT, так как перемещает процедурную логику внутрь машины SQL СУБД. Это заметно при добавлении данных больших объемов. Кроме того, обновление таблиц выполняется применительно к одному состоянию БД, а потому логически не сводимо к последовательному выполнению команд INSERT.

7.1.3.1. Неочевидный пример: многократные добавления в одну и ту же таблицу

Вот пример использования формально многотабличного оператора INSERT для занесения данных в БД с попутным выполнением переформатирования (похожее переформатирование, но только в запросе SELECT, а не при добавлении в таблицы БД может с версии 11 выполняться конструкцией SELECT ... FROM ... UNPIVOT ...).

Построим таблицу исходных данных:

```
CREATE TABLE e1 AS SELECT ename, sal, comm FROM emp;
```

Построим пустую таблицу для результата, где вместо двух полей SAL и COMM каждому сотруднику сопоставим отдельное поле с указанием вида платежа и отдельное поле для величины платежа:

```
CREATE TABLE e2 ( ename, amount )
AS
  SELECT ename, sal FROM emp WHERE 1 = 2
;

ALTER TABLE e2
  ADD (
    payment VARCHAR2 ( 10 ) CHECK ( payment IN ( 'salary', 'commission' ) )
  );
```

Теперь заполнение таблицы E2 можно выполнить следующим образом:

```
INSERT ALL
  WHEN sal IS NOT NULL THEN INTO e2 VALUES ( ename, sal, 'salary' )
  WHEN comm IS NOT NULL THEN INTO e2 VALUES ( ename, comm, 'commission' )
SELECT * FROM e1
;
```

Результат:

ENAME	AMOUNT	PAYMENT
SMITH	800	salary
ALLEN	1600	salary
WARD	1250	salary
JONES	2975	salary
MARTIN	1250	salary
BLAKE	2850	salary
CLARK	2450	salary
SCOTT	3000	salary
KING	5000	salary
TURNER	1500	salary
ADAMS	1100	salary
JAMES	950	salary
FORD	3000	salary
MILLER	1300	salary
ALLEN	300	commission

```
WARD          500 commission
MARTIN        1400 commission
TURNER        0 commission
```

18 rows selected.

7.1.3.2. Неочевидный пример: добавления в связанные таблицы

Если соблюдать правильную последовательность проверок условий, то многотабличная вставка позволяет одним оператором INSERT добавлять данные в связанные таблицы.

Положим, что данные для добавления в БД приходят в формате, показанном следующими примерами:

```
SELECT 1111 empno, 'QUEEN' ename, 20 deptno,      NULL dname FROM dual;
SELECT 2222 empno, 'DUKE'   ename, 50 deptno, 'JANITARY' dname FROM dual;
```

Считается, что если в пришедших данных название отдела отсутствует (первая строка), то нужно добавить только сотрудника в EMP, а в противном случае (вторая строка) добавить и запись о новом отделе в DEPT, и о новом сотруднике в EMP. Это делается, но тонкость в том, что две таблицы связаны правилом внешнего ключа, и поэтому первой должна идти проверка на добавление отдела, например:

```
INSERT ALL
  WHEN dname IS NOT NULL THEN          -- если требуется, сначала добавим отдел ...
    INTO dept ( deptno, dname ) VALUES ( deptno, dname )
  WHEN empno IS NOT NULL THEN          -- и во вторую очередь сотрудника ...
    INTO emp  ( empno, ename, deptno ) VALUES ( empno, ename, deptno )
-- добавляемые строки:
SELECT 1111 empno, 'QUEEN'  ename, 20 deptno,      NULL dname FROM dual
UNION ALL
SELECT 2222 empno, 'DUKE'   ename, 50 deptno, 'JANITARY' dname FROM dual
;
```

Проверка «табельного номера» EMPNO выполняется на всякий случай, и обеспечивает третий вариант: добавление одного только отдела, когда не указаны сведения о сотруднике.

Результатом станет добавление трех строк: двух в EMP и одной в DEPT.

Упражнение. Выполнить добавление данных как выше, удостовериться в появлении новых записей в таблицах EMP и DEPT, и вернуть таблицам прежние данные, выдав ROLLBACK. Поменять две проверки местами и наблюдать реакцию СУБД.

7.2. Изменение существующих значений полей строк

Логически операция UPDATE вторична, так как сводима к последовательности DELETE и INSERT, но в системах SQL технически не отрабатывается. (Строго говоря это не совсем так. Технически Oracle способна в некоторых случаях именно удалить запись в БД, представляющую строку таблицы, и добавить вместо нее новую, но это не единственный способ осуществления операции.) Она используется ради удобства применения. Любопытно, что если изменяется поле индексированного столбца и заодно с данными таблицы изменяется индекс, его изменение осуществляется ровно последовательным удалением из индекса старого значения и добавлением нового.

Примеры употребления:

```
UPDATE proj SET pname = 'GAMMA' WHERE projno = 30;
```

```
UPDATE proj
SET
  pname = 'GAMMA'
, budget = budget * 1.05
WHERE projno = 30
;
```

Поскольку речь идет об изменении значений полей уже существующих строк, в предложении UPDATE присутствует фраза WHERE, уточняющая множество строк для внесения изменения. Правила записи фразы WHERE те же, что и для предложения SELECT. Подобно как в SELECT, если в предложении UPDATE фраза WHERE не указана, изменение коснется всех строк источника данных.

Следующая форма UPDATE предполагает указание списка проставляемых в таблицу значений только однострочным подзапросом:

```
UPDATE proj
SET
  ( pname, budget ) =
  ( SELECT pname || '1', budget * 0.95 FROM dual )
;
```

Упражнение. Проверить, каков будет результат, если

- вложенный SELECT вернет более одной строки
- вложенный SELECT не вернет не одной строки
- столбец BUDGET будет незаполнен (NULL)
- столбец BUDGET будет частично заполнен

Еще пример формулирования предложения UPDATE:

```
UPDATE proj
SET budget =
  CASE
    WHEN pname = 'GAMMA' THEN budget
    WHEN budget IS NULL THEN 0
  ELSE NULL
END
;
```

На деле это всего лишь пример использования оператора CASE в построении выражения.

Операция UPDATE изменения существующих значений — множественная в силу своей формулировки.

7.3. Общие свойства INSERT и UPDATE

Операции INSERT и UPDATE роднит то, что обе по сути выполняют присвоение значений. Далее говорится о связанных с этим общими их свойствами.

7.3.1. Использование умолчательных значений в INSERT и UPDATE

Выражение для значения поля добавляемой или изменяемой строки можно заменить словом DEFAULT (разрешено в SQL:1999). В случае, когда в определении столбца присутствует выражение для вычисления умолчательного значения, именно оно и будет вычислено, и результат занесен в поле. Если умолчательное значение столбца явно не задавалось, указание слова DEFAULT в качестве значения равносильно указанию NULL (можно полагать, что если в определении столбца конструкция DEFAULT явно не указана, молчаливо предполагается DEFAULT NULL).

Пример:

```
CREATE TABLE t ( r NUMBER, a NUMBER ( 3 ), b NUMBER DEFAULT 123 )
;
INSERT INTO t ( r, a, b ) VALUES ( 1, 1, 2 );
INSERT INTO t ( r, a, b ) VALUES ( 2, NULL, NULL );
INSERT INTO t ( r, a, b ) VALUES ( 3, DEFAULT, DEFAULT );
```

```
INSERT INTO t ( r          ) VALUES ( 4 );
INSERT INTO t          VALUES ( 5, DEFAULT, DEFAULT );
```

Проверка:

```
SQL> SELECT * FROM t;
```

R	A	B
1	1	2
2		
3		123
4		123
5		123

7.3.2. Аномалия проверки занесенного в БД значения

Необычное поведение традиционных операций сравнения ($=$, $<$ и др.) с NULL влечет непривычный эффект проверки добавленного в БД командами INSERT и UPDATE значения. Обратимся снова к таблице T из предыдущего примера:

```
SQL> VARIABLE n NUMBER
SQL> INSERT INTO t ( r, a ) VALUES ( 6, :n );
```

1 row created.

```
SQL> SELECT * FROM t WHERE r = 6 AND a = :n;
```

no rows selected

```
SQL> SELECT COUNT ( * ) FROM t WHERE r = 6;
```

COUNT(*)
1

Такое поведение особенно неприятно в программе, и для привлечения внимания именно к этому контексту употребления вместо явного упоминания NULL в примере была применена переменная SQL*Plus. При простом объявлении переменной N она не получила никакого значения, так что появление NULL в запросах оказалось скрытым с ее помощью.

Можно вспомнить, что корни такого поведения Oracle уходят в стандарт SQL.

Другая аномалия диалекта Oracle SQL не связана со стандартом, и вызвана автоматическим приведением заносимого значения к типу столбца. Продолжим вышеприведенный пример:

```
SQL> EXECUTE :n := 1.8
SQL> UPDATE t SET a = :n WHERE r = 6;
```

1 row updated.

```
SQL> SELECT * FROM t WHERE r = 6 AND a = :n;
```

no rows selected

```
SQL> SELECT a FROM t WHERE r = 6;
```

A
2

7.4. Удаление строк из таблицы

7.4.1. Выборочное удаление

Основной оператор для удаления строк из таблицы — DELETE. Примеры:

```
DELETE FROM proj WHERE projno = 16;

DELETE FROM proj WHERE pname IS NULL;

DELETE FROM dept_copy;
```

Поскольку удаляться могут только существующие строки, ради их указания в предложении DELETE присутствует в общем случае фраза WHERE.

Операция удаления строк DELETE — множественная в силу своей формулировки.

7.4.2. Вариант полного удаления

Вместо полного удаления строк командой DELETE (в отсутствии фразы WHERE), например, вместо:

```
DELETE FROM dept_copy;
```

можно употреблять более быструю команду TRUNCATE TABLE:

```
TRUNCATE TABLE dept_copy;
```

Особенности TRUNCATE TABLE:

- DDL- операция → невосстанавливаемая операция;
- быстро выполняется (ощутимо на больших таблицах), поскольку строки удаляются как результат укорачивания структуры хранения данных таблицы в БД («сегмента»), а не поштучно, как при DELETE.

Если очищенную от строк таблицу предполагается впоследствии снова заполнять, время последующего заполнения сократится, если выдать:

```
TRUNCATE TABLE dept_copy REUSE STORAGE;
```

В этом случае строки будут полагаться удаленными, а структура хранения данных таблицы в БД останется внешне неизменным.

7.5. Объединение INSERT, UPDATE и DELETE в одном операторе

В версии СУБД 9 появилась команда MERGE, позволяющая либо изменить существующие строки какой-нибудь таблицы, либо добавить — в зависимости от сформулированного условия. Условие формулируется на основе соединения целевой таблицы с другой (в общем случае — с источником данных, в качестве которого, однако, помимо обычной таблицы могут выступать еще представление данных, таблицы с внешним и хранением с временным хранением данных, а также подзапрос).

Заполним таблицу BONUS данными о сотрудниках, положим, имеющих комиссионные:

```
INSERT INTO bonus
  SELECT ename, job, sal, comm
  FROM emp
  WHERE comm IS NOT NULL
;
```


Теперь обновим BONUS данными, «поступившими» из таблицы EMP. Если сотрудник из EMP уже есть в BONUS, повысим ему зарплату, а если нет — добавим к списку BONUS:

```
MERGE
INTO bonus b
USING emp e
ON ( b.ename = e.ename )
WHEN MATCHED THEN
    UPDATE SET b.sal = e.sal * 10
WHEN NOT MATCHED THEN
    INSERT VALUES ( e.ename, e.job, e.sal, e.comm )
;
```

Одна из двух проверок формально может и отсутствовать, что позволяет всего одной командой SQL просто добавить записи, отсутствующие в однотипной таблице.

В версии 10 фраза во фразе WHEN MATCHED можно дополнительно указать DELETE, например:

```
MERGE
INTO bonus b
USING emp e
ON ( b.ename = e.ename )
WHEN MATCHED THEN
    UPDATE SET b.sal = b.sal / 10
    DELETE WHERE b.sal < 1000
;
```

(Вернули BONUS в состояние до первого оператора MERGE).

Назначение операции MERGE — ускорить обновление больших таблиц. Обратите внимание, что обновление выполняется применительно к одному состоянию БД и поэтому не логически сводимо к последовательному выполнению команд INSERT, UPDATE и, возможно, DELETE.

7.6. Целостность выполнения операторов обновления данных и реакция на ошибки

В процессе выполнения множественных операторов обновления данных таблиц изменение отдельной строки может вызвать ошибку — например, вылиться в попытку нарушить ограничение целостности. В таких случаях операция обновления прекращается, а изменения, успевшие произойти до возникновения проблемы, отменяются. Иными словами, операции DML обновления данных либо выполняются целиком, либо, в конечном счете, не выполняются вовсе.

7.6.1. Реакция на ошибки в процессе исполнения

Традиционная реакция на ошибки в процессе выполнения изменяющего данные оператора («все или ничего») логически оправдана, но не всегда практична в случае больших объемов данных. В версии 10.2 введена возможность не отказываться от исполнения огульно, а вместо этого запоминать возникающие на отдельных строках ошибки в специально подготовленной таблице для последующего разбирательства. Специальную таблицу можно завести с помощью особой системной процедуры. Пример ее создания для таблицы EMP и дальнейшего употребления приводится ниже.

```
SQL> EXECUTE DBMS_ERRLOG.CREATE_ERROR_LOG ( 'EMP', 'ERR_EMP' )

PL/SQL procedure successfully completed.

SQL> INSERT INTO emp ( empno ) VALUES ( 1111 );

1 row created.
```

```

SQL> /
INSERT INTO emp ( empno ) VALUES ( 1111 )
*
ERROR at line 1:
ORA-00001: unique constraint (SCOTT.PK_EMP) violated

SQL> INSERT INTO emp ( empno ) VALUES ( 1111 )
      2 LOG ERRORS INTO err_emp ( 'today error' ) REJECT LIMIT 10;

0 rows created.

SQL> COLUMN ora_err_mesg$ FORMAT A45 WORD
SQL> COLUMN ora_err_tag$ FORMAT A12
SQL> COLUMN err# FORMAT 99999
SQL> COLUMN empno FORMAT A6
SQL> SELECT ora_err_number$ err#, ora_err_mesg$, ora_err_tag$, empno
      2 FROM err_emp;

ERR# ORA_ERR_MESG$ ORA_ERR_TAG$ EMPNO
-----
1 ORA-00001: unique constraint (SCOTT.PK_EMP) today error 1111
violated

```

Обратите внимание, что второй оператор INSERT строку не добавляет (правило первичного ключа нарушить нельзя), но и ошибку в программу не возвращает. Если же ошибок окажется более, чем указано в конструкции REJECT LIMIT, выполнение оператора все-таки будет прервано с реакцией в виде общей ошибки как обычно.

7.6.2. Особая реакция на ошибки нарушения уникальности при выполнении оператора INSERT

В версии 11.2 предусмотрена особая реакция на попытки нарушения уникальности индекса (в частности, нарушения правил уникальной группы столбцов и первичного ключа) при добавлении данных оператором INSERT. Пример того, как это работает:

```

SQL> CREATE TABLE losers AS SELECT * FROM emp WHERE 1 = 2;

Table created.

SQL> ALTER TABLE losers ADD PRIMARY KEY ( empno );

Table altered.

SQL> INSERT INTO losers SELECT * FROM emp WHERE deptno IN ( 10, 30 );

9 rows created.

SQL> INSERT INTO losers SELECT * FROM emp WHERE job = 'CLERK';
INSERT INTO losers SELECT * FROM emp WHERE job = 'CLERK'
*
ERROR at line 1:
ORA-00001: unique constraint (SCOTT.SYS_C0015038) violated

SQL> INSERT /*+ IGNORE_ROW_ON_DUPKEY_INDEX ( losers ( empno ) ) */
      2 INTO losers
      3 SELECT * FROM emp WHERE job = 'CLERK'
      4 ;

2 rows created.

```

Очевидно, что попытка добавить строку со значением, нарушающим уникальность индекса, приводит не к ошибке выполнения всего оператора, а лишь к отказу от приема в БД нарушающей строки.

Нестандартное поведение INSERT обеспечивается необычным же образом: с помощью подсказки. Подсказки в Oracle — это псевдокомментарии особого вида, добавляемые после слов SELECT, INSERT, UPDATE или DELETE. Обычно они используются программистом для воздействия на оптимизатор запросов СУБД с целью получить ответ быстрее или менее затратно (и в этом качестве воспринимается многими специалистами неоднозначно). В случае же

с IGNORE_ROW_ON_DUPKEY_INDEX мы имеем дело с редкой подсказкой, оказывающей влияние не только на способ вычисления, но и на результат. Не совсем ясно, почему фирма Oracle прибегла к такому странному оформлению действия, и можно предположить, что в будущих версиях СУБД на смену этой подсказке придет синтаксическая конструкция.

Вероятное применение такой разновидности INSERT может лежать в технологии формирования рабочих массивов данных.

7.7. Запрет на изменение данных в таблице

С версии 11 изменение данных в таблице можно запретить, переведя таблицу в состояние READ ONLY:

```
ALTER TABLE emp READ ONLY;
```

До этой версии запретить изменения можно было только одновременно во всех объектах, хранящих свои данные в конкретном табличном пространстве.

7.8. Фиксация или отказ от изменений в БД

Для сеанса, выдающего команды DML на изменения данных, СУБД создает видимость, что они выполняются сразу в БД. Выполнив тут же запрос, пользователь увидит, будто данные изменились. На деле же они попадут в базу только после выдачи сеансом специальной команды фиксации изменений. Только после этого они станут видны прочим сеансам.

Все изменения со стороны индивидуальных команд DML заносятся Oracle в БД только группами, в рамках транзакции, по завершению транзакции. Команды завершения текущей транзакции:

```
COMMIT [WORK];
```

```
ROLLBACK [WORK];
```

Завершение транзакции с *фиксацией* изменений, внесенных операторами DML, происходит только по выдаче (а) команды COMMIT или (б) оператора DDL (скрытым образом завершающего свои действия по изменению таблиц словаря-справочника той же командой COMMIT).

Завершение транзакции с *отменой* изменений, внесенных было операторами DML, происходит только по выдаче команды ROLLBACK, которая или явно выдается программистом, или, в некоторых случаях, неявно порождается самой СУБД (например, по результату аварийного останова работы программы из-за неперехваченной исключительной ситуации).

Упражнение. Вставить запись в имеющуюся таблицу — откатить изменения. Вставить запись — зафиксировать изменения. Создать таблицу, вставить запись — откатить изменения. Сохранилась ли таблица и ее данные? Создать заполненную таблицу предложением CREATE TABLE ... AS SELECT Откатить изменения.

Oracle нумерует внесение изменений в БД сквозным образом постоянно растущими номерами, называемыми SCN (system change number, номер изменения системы). Каждая команда COMMIT переводит БД в новое состояние, характеризуемое новым номером SCN. Данные БД в более ранних состояниях становятся после этого доступны только средствами (а) восстановления по резервным копиям, или (б) «быстрого» восстановления (flashback).

В версии 11 Oracle разрешила не только анулировать командой ROLLBACK изменения, совершавшиеся в рамках завершаемой транзакции, но также и отменять изменения, выполнявшиеся по очереди несколькими последними транзакциями (завершенными ранее командой COMMIT). Но делается это уже не операцией SQL, а программно, средствами системного пакета DBMS_FLASHBACK.

7.8.1. Данные о номере последней транзакции, изменившей строку таблицы

В версии 10 стало возможным с помощью системной переменной («псевдостолбца») ORA_ROWSCN узнать SCN транзакции, внесшей последнее изменение в строку. Если ничего не предпринимать специально, то этот номер будет приближенный и соответствовать фактически не строке, а блоку, в котором хранится в БД строка. (Сама возможность хранить вместе со строкой SCN ее последней правки были и раньше и использовалась для параллельной репликации, но смотреть этот номер в программе было нельзя). Пример:

```
SQL> CREATE TABLE dscn AS SELECT * FROM dept;
```

Table created.

```
SQL> UPDATE dscn SET dname = LOWER ( dname ) WHERE ROWNUM <= 2;
```

2 rows updated.

```
SQL> SELECT dname, ora_rowscn FROM dscn;
```

DNAME	ORA_ROWSCN
accounting	3812962
research	3812962
SALES	3812962
OPERATIONS	3812962

```
SQL> COMMIT;
```

Commit complete.

```
SQL> SELECT dname, ora_rowscn FROM dscn;
```

DNAME	ORA_ROWSCN
accounting	3812999
research	3812999
SALES	3812999
OPERATIONS	3812999

Однако если создать таблицу с особым указанием, в ней появится скрытый столбец (длиною 6 байтов), рассчитанный на хранение номера SCN индивидуально для каждой строки:

```
SQL> DROP TABLE dscn;
```

Table dropped.

```
SQL> CREATE TABLE dscn ROWDEPENDENCIES AS SELECT * FROM dept;
```

Table created.

```
SQL> UPDATE dscn SET dname = LOWER ( dname ) WHERE ROWNUM <= 2;
```

2 rows updated.

```
SQL> SELECT dname, ora_rowscn FROM dscn;
```

DNAME	ORA_ROWSCN
accounting	
research	
SALES	3814027
OPERATIONS	3814027

```
SQL> COMMIT;
```

Commit complete.

```
SQL> SELECT dname, ora_rowscn FROM dscn;
```

DNAME	ORA_ROWSCN
-------	------------

accounting	3814035
research	3814035
SALES	3814027
OPERATIONS	3814027

Эти данные можно использовать в программе, чтобы определить, изменялась ли строка с какого-то времени. Перевести SCN в астрономическое время (приблизенно, с версии 10 плюс — минус 3 секунды) можно обращением к специальной функции, например:

```
SELECT dname, SCN_TO_TIMESTAMP ( ora_rowscn ) FROM dscn;
```

7.9. Обращение с прошлыми данными после внесения изменений

Традиционно все существующие СУБД создавались как средства моделирования текущего состояния предметной области. Выдача COMMIT переводит БД в новое состояние, и предыдущие данные оказываются потерянными. Такое поведение проще программировать разработчикам СУБД, но оно не всегда удобно пользователям. В частности:

- для восстановления данных, потерянных в результате непродуманной выдачи COMMIT, пусть даже совсем недавно, приходилось довольствоваться восстановлением по резервной копии БД, что долго и требует наличия собственно резервной копии;
- нередко возникающую потребность моделировать историю изменения данных приходится имитировать разработчику приложения на свое усмотрение.

В версии 9 в Oracle открылись возможности получать от СУБД значения данных таблицы по состоянию на прошлый момент, невзирая на осуществлявшиеся за это время операции фиксации транзакций. В версии 10 эти возможности получили свое развитие. Техническая основа у них разная: использование временно сохранившихся данных в табличном пространстве типа UNDO; использование свободного места в табличном пространстве БД с данными таблицы; особый способ журнализации данных. Отсюда проистекают различия в доступных сроках давности для восстановления в разных случаях.

7.9.1. Обращение к прошлым значениям данных в таблице

Для (быстрого) запроса к прежним данным ссылку на таблицу во фразе FROM предложения SELECT следует сопроводить указанием конструкции AS OF. Пример (выполнить в качестве упражнения последовательно):

```
DELETE FROM emp;

COMMIT;

SELECT * FROM emp;

SELECT *
FROM   emp AS OF TIMESTAMP ( SYSTIMESTAMP - INTERVAL '1' MINUTE )
;

INSERT INTO emp
SELECT *
FROM   emp AS OF TIMESTAMP ( SYSTIMESTAMP - INTERVAL '1' MINUTE )
;

SELECT * FROM emp;
```

Синтаксис допускает употребление скобок в запросе выше, но не требует этого. Уточнение конструкцией AS OF допускается одновременно у нескольких источников данных:

```
SELECT e.ename, d.dname
FROM
    emp AS OF TIMESTAMP ( SYSTIMESTAMP - INTERVAL '15' MINUTE ) e
  INNER JOIN
    dept AS OF TIMESTAMP ( SYSTIMESTAMP - INTERVAL '30' MINUTE ) d
  USING ( deptno )
;
```

На момент в прошлом можно сослаться не только временем (TIMESTAMP), но и номером SCN состояния БД.

Подобное извлечение старых значений возможно, однако, только за период, когда к таблице не применялись команды DDL изменения ее описания.

Давность воспроизводимых данных ограничивается размером свободного места в табличном пространстве UNDO, которое определяется (а) интенсивностью изменений БД и (б) полным размером пространства.

Версия 10 пошла еще дальше, разрешив вместо AS OF уточнять имя таблицы во фразе FROM конструкцией VERSIONS BETWEEN, позволяющей извлекать из БД *историю изменения строк* за определенный период. Следующим примером можно продолжить приводившийся выше код:

```
COLUMN versions_endtime    FORMAT A22
COLUMN versions_starttime  FORMAT A22
SELECT
    empno
  , sal
  , versions_starttime
  , versions_endtime
  , versions_xid
  , versions_operation
FROM
emp VERSIONS BETWEEN TIMESTAMP MINVALUE AND MAXVALUE
;
```

Прочие поля строк таблицы EMP в этом запросе не выданы из экономии места.

Упражнение. Изменить несколько раз зарплаты разным сотрудникам и просмотреть историю изменений.

7.9.2. Восстановление данных существующих таблиц, ранее удаленных таблиц и всей БД

С версии 10 открылась возможность одной командой восстановить *строки* таблицы на момент в прошлом. Это позволяет вместо вышеуказанного INSERT ... SELECT ... AS OF записать:

```
FLASHBACK TABLE emp
TO TIMESTAMP ( SYSTIMESTAMP - INTERVAL '1' MINUTE )
;
```

Тем не менее:

- (а) эти две команды не равносильны, когда перед восстановлением строк таблица была не пуста;
- (б) чтобы восстановление строк таблицы командой FLASHBACK было возможным, мы должны разрешить системе изменять физические адреса строк таблицы. Например, выдать: ALTER TABLE emp ENABLE ROW MOVEMENT;

Для полного восстановления *таблицы*, ранее удалявшейся, команда FLASHBACK выглядит иначе. Вариант простого восстановления (из мусорной корзины):

```
FLASHBACK TABLE emp TO BEFORE DROP;
```

Вариант с переименованием (если с момента удаления таблицы EMP ее имя оказалось захвачено другим объектом, или по иным причинам):

```
FLASHBACK TABLE emp TO BEFORE DROP RENAME TO emp_old;
```

Если таблица переименовывалась несколько раз, вместо имени таблицы можно подставить внутреннее название «удаленной» таблицы, полученное из системной таблицы USER_RECYCLEBIN.

Технически такая возможность достигается сохранением старой структуры хранения данных («сегмента») таблицы в табличном пространстве при выполнении DROP TABLE, что определяет границы применимости такого подхода.

Если же *база данных* работает в специальном режиме flashback, команда FLASHBACK позволяет восстанавливать ее целиком:

```
FLASHBACK DATABASE TO TIMESTAMP SYSTIMESTAMP - INTERVAL '1' HOUR;
```


8. Объявленные ограничения целостности

Все величины, заносимые в таблицу, обязаны входить во множества, допускаемые типами соответствующих столбцов. Ограничения целостности данных позволяют добавить для них требования, дополнительные к соблюдению типа. Объявленные (схемные, формальные, «декларативные») ограничения целостности записываются («провозглашаются») в виде условий, которые должны соблюдаться, *явно как таковые*, на уровне схемы данных, и этим отличаются от правил целостности, сформулированных в виде запрограммированных проверок (см. ниже). Поэтому иначе такие ограничения можно называть «явными». Оригинальный термин имеет полное название integrity data constraints — «ограничения на значения данных, налагаемые для более точного учета обстоятельств предметной области», но часто сокращается до integrity constraints или даже просто constraints. Слово integrity вряд ли хорошо понятно массам разработчиков.

Само понятие объявленных ограничений целостности в SQL было унаследовано от реляционной модели и усложнилось вместе с развитием стандарта. В Oracle номенклатура ограничений целостности в целом соответствует SQL-92 (при том, что объем реализации не выдержан), но не доведена до уровня SQL:1999. Так, Oracle не позволяет завести ограничение целостности на уровне БД (с помощью служебного слова ASSERTION) и сильно ограничен в формулировании условия проверки значений конструкцией CHECK тем, что не допускает обращения к данным базы.

Слово ASSERTION из стандарта SQL подсказывает еще один перевод (и понимание) integrity constraints, как «утвердительные ограничения целостности».

- отдельного поля строки в таблице,
- отдельной строки,
- пары таблиц.

Проверка на выполнение действующих объявленных ограничений целостности выполняется СУБД автоматически и всегда, вне зависимости от источника поступления изменений, чем и гарантировано их соблюдение, в отличие, скажем, от проверок вводимых значений, осуществляемых клиентскими прикладными программами.

Oracle позволяет формулировать подобные ограничения при создании таблицы командой CREATE TABLE, а для уже существующих таблиц их можно добавлять и отменять командами:

- ALTER TABLE ... MODIFY — добавление ограничений всех видов и снятие ограничения NOT NULL;
- ALTER TABLE ... ADD/DROP — добавление и снятие ограничений всех видов кроме NOT NULL.

Всем ограничениям целостности, сформулированными в схеме, Oracle сообщает имена. Если при создании ограничения употребить конструкцию CONSTRAINT *имя*, ограничение получит имя от программиста, в противном случае СУБД создаст имя по своему усмотрению. Сведения о каждом существующем ограничении можно найти в таблице словаря-справочника USER_CONSTRAINTS по его имени. Неудачное имя ограничения можно изменить; к примеру:

```
ALTER TABLE projx RENAME CONSTRAINT sys_c0011509 TO name_is_needed;
```

Сведения же о столбцах, участвующих в правилах ограничений, можно найти в таблице USER_CONS_COLUMNS (иногда потребуется заглянуть в ALL_CONS_COLUMNS).

8.1. Разновидности объявленных ограничений целостности

8.1.1. Ограничение NOT NULL

Ограничение NOT NULL обязывает столбец или группу столбцов всегда иметь значение (если группа — то хотя бы в одном поле). Требование непустоты столбца крайне желательно, так как избавляет

программиста от многочисленных забот, связанных с особенностями обработки NULL. К сожалению, требования предметной области и некоторые действия в SQL (например, GROUP BY ROLLUP ...) не позволяют совсем отказаться от столбцов со свойством NULL.

Это единственное из ограничений целостности, информация о котором хранится не только в таблице USER_CONSTRAINTS, но и в таблице USER_TAB_COLUMNS в качестве свойства столбца. (Когда-то признак NULL/NOT NULL формально считался свойством столбца, а не ограничением целостности). По этой причине добавление и упразднение этого ограничения оформляется по правилам изменения свойства столбца, только через ключевое слово MODIFY:

```
ALTER TABLE proj MODIFY ( budget NOT NULL );  
-- создание ограничения с системным именем; скобки необязательны
```

```
ALTER TABLE proj MODIFY ( budget NULL );  
-- упразднение ограничения; скобки необязательны
```

```
ALTER TABLE proj MODIFY ( budget CONSTRAINT is_mandatory NOT NULL );  
-- создание ограничения с именем, заданным программистом
```

Имеется случай (версии 12+), когда ограничение NOT NULL навязывается столбцу неявно. Это когда столбцу приписывается умолчательное значение, и дополнительно уточняется, что оно же появится в столбце при попытке «добавить» туда NULL:

```
ALTER TABLE dept_copy MODIFY ( cat DEFAULT ON NULL 10 );
```

В современных версиях Oracle самостоятельное ограничение NOT NULL будет оформлено технически как ограничение вида CHECK с условием для проверки: budget IS NOT NULL и одновременно будет зафиксировано в USER_CONSTRAINTS значением NULLABLE = 'Y'. Свойство NOT NULL, вытекающее из правила первичного ключа, будет отражено только в USER_CONSTRAINTS.

8.1.2. Первичные ключи

От столбцов, назначенных первичным ключом, требуется, чтобы значения в их полях всех строк были уникальными и имелись всегда (для ключа из нескольких столбцов значение должно быть хотя бы в одном поле). Примеры создания и удаления:

```
ALTER TABLE proj ADD PRIMARY KEY ( projno, pname );  
-- создание ограничения (первичный ключ на основе двух столбцов) с системным именем
```

```
ALTER TABLE proj DROP PRIMARY KEY;  
-- упразднение ограничения
```

```
ALTER TABLE proj ADD CONSTRAINT pk_proj PRIMARY KEY ( projno );  
-- создание ограничения с именем, заданным программистом
```

Значения в полях первичного ключа должны существовать всегда.

Некоторые типы столбцов не допускаются до формирования первичного ключа (например, LOB или TIMESTAMP WITH TIME ZONE).

8.1.3. Уникальность значений в столбцах

От столбцов, назначенных уникальными, требуется, чтобы значения в их полях всех строк были уникальными. Уникальность в SQL наиболее близка к понятию «альтернативного», «возможного» (candidate) или же просто «ключа» в реляционной модели.

Пример создания:

```
ALTER TABLE proj ADD UNIQUE ( pname );
```

Обратите внимание, что в столбце PNAME не запрещаются пропуски значений. По стандарту SQL уникальность отслеживается для *имеющихся* значений столбца. Если на такой столбец дополнительно наложить ограничение:

```
ALTER TABLE proj MODIFY ( pname NOT NULL );
```

он сможет играть роль ключа в реляционной модели и быть объявлен первичным (путем замены двух ограничений: UNIQUE и NOT NULL на одно PRIMARY KEY). Если же уникальной объявляется группа столбцов, сообщить ей de facto свойства ключа придется, объявив NOT NULL на группу.

Другое отличие ограничения уникальности от первичного ключа в том, что первых в таблице может быть сформулировано несколько, а второе присутствует разве что в единственном числе. Oracle не препятствует объявлению уникальности не только непересекающихся групп столбцов, но даже и повторяющихся. Следующая цепочка команд *не* вызовет ошибок:

```
CREATE TABLE t ( a NUMBER, b NUMBER, c NUMBER );
```

```
ALTER TABLE t ADD CONSTRAINT ab UNIQUE ( a, b );
```

```
ALTER TABLE t ADD CONSTRAINT bc UNIQUE ( b, c );
```

```
ALTER TABLE t ADD CONSTRAINT ba UNIQUE ( b, a );
```

Потребовать в таблице EMP, чтобы в один и тот же отдел одновременно не принималось двух сотрудников в одной должности, можно следующим образом:

```
ALTER TABLE emp
  ADD CONSTRAINT no_duplicates
    UNIQUE ( deptno, job, hiredate )
;
```

Более того, Oracle не запретит включить в состав уникальной группы столбцы первичного ключа, в том числе все из них. Последнее в реляционной теории соответствует понятию «суперключа» и невозможно для ключа.

Однако точное повторение списка имен столбцов в новом определении приведет к ошибке (что довольно необычно логически и вызвано техническими причинами реализации):

```
ALTER TABLE t ADD CONSTRAINT xx UNIQUE ( a, b );
-- Ошибка !
```

8.1.4. Внешние ключи

Столбцы, объявленные внешним ключом, обязаны (а) ссылаться на однотипные столбцы из другой или той же таблицы при условии, что адресат — это первичный ключ или уникальная группа столбцов, и (б) принимать только существующие в данный момент в столбцах-адресатах значения. Пример создания:

```
ALTER TABLE proj ADD ( ldept NUMBER ( 2 ) )
;
ALTER TABLE proj ADD FOREIGN KEY ( ldept ) REFERENCES dept ( deptno )
;
```

По правилам внешнего ключа в столбце LDEPT *не* запрещаются пропуски значений. Стандарт SQL требует от СУБД проверки соответствия значениям в столбцах-адресатах таблицы только *имеющихся* значений внешнего ключа; иными словами, значения в полях внешнего ключа могут отсутствовать.

Внешних ключей в таблице может быть определено несколько. Например, при более тщательном моделировании примера «сотрудники — отделы» в дополнение к имеющемуся внешнему ключу DEPTNO таблицы EMP можно было бы объявить внешним ключом столбец JOB, заставив его ссылаться на отдельную таблицу с описаниями штатных должностей.

8.1.4.1. Выбор таблицы-адресата для внешнего ключа

Столбцам внешнего ключа не запрещено ссылаться на столбцы своей же таблицы:

```
ALTER TABLE emp
  ADD CONSTRAINT valid_manager
    FOREIGN KEY ( mgr ) REFERENCES emp ( empno )
;
```

В стандарте SQL такой внешний ключ называется рекурсивным.

Равным образом внешнему ключу разрешено ссылаться на столбцы таблицы из другой схемы. Только в этом случае потребуется иметь на таблицу из другой схемы привилегию REFERENCES:

```
CONNECT scott/tiger
-- соединились с СУБД как SCOTT

GRANT REFERENCES ON dept TO yard;
-- выдали право ссылаться внешним ключом на поля DEPT из схемы YARD

CONNECT yard/pass
-- соединились с СУБД как YARD

CREATE TABLE emp AS SELECT * FROM scott.emp;
-- создали таблицу EMP по образу одноименной в схеме SCOTT

ALTER TABLE emp
  ADD
  FOREIGN KEY ( deptno ) REFERENCES scott.dept ( deptno )
;
-- установили ссылку на таблицу из другой схемы
```

Обратите внимание, что привилегии на SELECT к таблице-адресату в случае нахождения последней в иной схеме не требуется.

Пример использования такой возможности — поддержка в разных схемах ссылок на справочные таблицы, собранные вместе в отдельную схему. Правда, при таком подходе внесение изменений в БД потребует дополнительного внимания.

8.1.4.2. Поведение СУБД при попытке удалить «родительскую» запись

Обычное ограничение типа «внешний ключ» запрещает СУБД удалять родительскую запись, если на нее существуют в данный момент ссылки:

```
DELETE FROM dept WHERE deptno = 10;
```

Однако можно смоделировать и иную реакцию СУБД, разрешив-таки удаление родительской записи.

Указание ON DELETE CASCADE в определении ключа приведет заодно с удалением родительской записи к автоматическому удалению подчиненных *записей*:

```
CREATE TABLE x ( a NUMBER PRIMARY KEY );
CREATE TABLE y ( b NUMBER PRIMARY KEY,
```

```

c NUMBER REFERENCES x ( a ) ON DELETE CASCADE );

INSERT INTO x VALUES ( 1 );
INSERT INTO y VALUES ( 2, 1 );
DELETE FROM x;
SELECT * FROM y;

```

Обе таблицы пусты.

При наличии цепочки так определенных внешних ключей автоматическое удаление будет распространяться по цепочке:

```

CREATE TABLE z ( d NUMBER PRIMARY KEY,
e NUMBER REFERENCES y ( b ) ON DELETE CASCADE );

INSERT INTO x VALUES ( 1 );
INSERT INTO y VALUES ( 2, 1 );
INSERT INTO z VALUES ( 3, 2 );
DELETE FROM x;
SELECT * FROM z;

```

Автоматическим удалением по цепочке следует пользоваться с осторожностью.

Указание ON DELETE SET NULL в определении ключа приведет заодно с удалением родительской записи к автоматическому удалению значений в полях-ссылках подчиненных записей:

```

CREATE TABLE w ( f NUMBER REFERENCES z ( d ) ON DELETE SET NULL );

INSERT INTO z VALUES ( 3, NULL );
INSERT INTO w VALUES ( 3 );
DELETE FROM z;
SELECT * FROM w;

```

Строка в таблице Z пропала, а в таблице W осталась (проверьте!).

Обратите внимание, что фраза CASCADE CONSTRAINTS в предложении DROP TABLE не соответствует ни первому, ни второму из вышеприведенных вариантов, попросту удаляя ограничение типа «внешний ключ» и не трогая значений подчиненных записей:

```

INSERT INTO x VALUES ( 1 );
INSERT INTO y VALUES ( 2, 1 );
DROP TABLE x CASCADE CONSTRAINTS;
SELECT * FROM y;

DROP TABLE y CASCADE CONSTRAINTS;
DROP TABLE z CASCADE CONSTRAINTS;
DROP TABLE w CASCADE CONSTRAINTS;

```

Стандарт SQL дает право задавать аналогичное поведение СУБД при попытках изменить родительскую запись, используя для этого формулировку ON UPDATE CASCADE. Oracle такой возможности не дает. Мнения специалистов по поводу целесообразности подобной формулировки разделились на противоположные.

8.1.5. Дополнительное условие для занесения значений в поля строки

Этот вид ограничения записываются с использованием ключевого слова CHECK. Он позволяет сформулировать в форме условного выражения дополнительные проверки на заносимые в поля строки значения. Если условное выражение окажется ложным, СУБД отвергнет изменение значений (реляционная теория предпочитает соблюдать истинность условия, а не отсутствие нарушения, что не одно и то же). Пример:

```
ALTER TABLE dept_copy ADD CHECK (
    REGEXP_LIKE ( moody_rating, '^ (A{1,3}|B{1,3}|C) (\+|\-|\$)' )
);
```

Другие примеры возможной дополнительной проверки в описании столбца:

```
...
, zip CHAR ( 6 ) CHECK ( REGEXP_LIKE ( zip, '^[:digit:]+$' ) ) )
, ename VARCHAR2 ( 30 ) CHECK ( ename = INITCAP ( ename ) )
, empno NUMBER ( 4 ) CHECK ( empno BETWEEN 1000 AND 2000 )
, educ CLOB CHECK ( educ IS JSON ) /* Oracle 12+ */
...
```

Пример указания условных проверок при создании таблицы:

```
CREATE TABLE emp1
(
    empno NUMBER
, job VARCHAR2 ( 9 ) DEFAULT 'SALESMAN'
, sal NUMBER ( 10, 2 )
, comm NUMBER ( 9, 2 )
,
    CONSTRAINT pk_emp1 PRIMARY KEY ( empno )
, CONSTRAINT sal_low_limit CHECK ( sal >= 500 )
, CONSTRAINT commission_positive CHECK ( comm >= 0 )
, CONSTRAINT emp_number_strictly_positive CHECK ( empno > 0 )
, CONSTRAINT check_whole_earning CHECK ( sal + comm < 7000 )
, CONSTRAINT has_commission CHECK ( job = 'SALESMAN' OR comm IS NULL )
);
```

Упражнение. Попробовать выполнить:

```
INSERT INTO emp1 ( empno, sal, comm ) VALUES ( 1, 500, 1000 );
INSERT INTO emp1 ( empno, sal, comm ) VALUES ( 2, 8000, 1000 );
INSERT INTO emp1 ( empno, sal ) VALUES ( 3, 8000 );
```

Как нужно изменить определение таблицы, чтобы исправить ситуацию и сообщить СУБД интуитивное понимание правила $SAL + COMM < 7000$? Решить аналогичную проблему для ограничения HAS_COMMISSION.

Последний пример из упражнения подчеркивает, что условное выражение в проверке CHECK обрабатывается отлично от фразы WHERE (или оператора CASE). В случае CHECK изменение отвергается, когда условное выражение дает FALSE; в случае WHERE строка выбирается для обработки, когда условное выражение дает TRUE. Область расхождения в трактовке — NULL в качестве результата оценки логического выражения.

Обратите внимание, что формулировать ограничения целостности в тексте предложения CREATE TABLE допускается как на уровне столбца (если только ограничение касается единственного столбца, как, например, PK_EMP1 или SAL_LOW_LIMIT), так и на уровне таблицы (CHECK_WHOLE_EARNING и HAS_COMMISSION). Такая разница в записи не отражается на свойствах таблицы.

Упражнение. Удалить созданную таблицу EMP1 и создать заново, но сформулировав те же ограничения целостности на уровне таблицы.

В отличие от стандарта SQL, в Oracle условное выражение в CHECK не имеет право содержать обращения к БД (в том числе через посредство функций), что существенно ослабляет его общую силу. Сверх того, запрещено в таких выражениях обращаться к псевдостолбцам, а также, в отличие от

выражений для конструкции DEFAULT, к системным переменным и к значениям контекста сеанса USERENV.

8.2. Добавление ограничения с отказом от проверки соответствия имеющимся данным

Ограничения типа «внешний ключ» и «проверка значений» (FOREIGN KEY и CHECK) могут добавляться к существующей таблице, даже если данные в таблице им противоречат. Отказаться от проверки данных при добавлении ограничения можно с помощью слова NOVALIDATE. В этом случае ограничение вступит в силу, однако не исключается, что ранее занесенные данные будут его нарушать. Автоматическая проверка ограничения, как и полагается, будет распространяться на все последующие попытки изменения данных в таблице:

```
SQL> CREATE TABLE deptc AS SELECT * FROM dept;

Table created.

SQL> INSERT INTO deptc ( deptno ) VALUES ( 31 );

1 row created.

SQL> ALTER TABLE deptc ADD CHECK ( deptno <= 30 );
ALTER TABLE deptc ADD CHECK ( deptno <= 30 )
*
ERROR at line 1:
ORA-02293: cannot validate (SCOTT.SYS_C005484) - check constraint violated

SQL> ALTER TABLE deptc ADD CHECK ( deptno <= 30 ) NOVALIDATE;

Table altered.

SQL> INSERT INTO deptc ( deptno ) VALUES ( 30 );

1 row created.

SQL> INSERT INTO deptc ( deptno ) VALUES ( 32 );
INSERT INTO deptc ( deptno ) VALUES ( 32 )
*
ERROR at line 1:
ORA-02290: check constraint (SCOTT.SYS_C005484) violated

SQL> SELECT * FROM deptc WHERE deptno > 30;
```

DEPTNO	DNAME	LOC
40	OPERATIONS	BOSTON
31		

То есть ограничение на изменение данных действует, но в таблице встречаются его нарушения (возникшие ранее).

Такая возможность добавить ограничение без проверки соответствия ему имеющихся данных позволяет:

- сократить время в случае большой таблицы, когда у программиста есть уверенность, что имеющиеся данные ограничению не противоречат;
- включить ограничение немедленно для последующих операций с данными, отложив проверку ранее накопленных данных на потом.

8.3. Приостановка проверки объявленных ограничений в пределах транзакции

В соответствии со стандартом SQL ANSI/ISO, с версии Oracle 8.1 именованные объявленные ограничения целостности в Oracle можно определять со словом DEFERRABLE, например:

```
ALTER TABLE dept ADD CONSTRAINT u_names UNIQUE ( loc ) DEFERRABLE;
```

В этом случае станет возможным отложить автоматическую проверку ограничения на период до завершения текущей транзакции (подобная степень долговечности отмены проверки оправдывает употребление термина «приостановка»). С этой целью в сеансе связи с СУБД следует выдать:

```
SET CONSTRAINT u_names DEFERRED;
```

Теперь можно внести изменение, нарушающее ограничение:

```
INSERT INTO dept ( loc, deptno ) VALUES ( 'BOSTON', 50 );
```

Если дубликат не будет устранен, то сообщение об ошибке появится только при попытке выполнить COMMIT или же явочным путем возобновить конкретную проверку:

```
SET CONSTRAINT u_names IMMEDIATE;
```

Завершение транзакции (любым образом, COMMIT/ROLLBACK) отменяет *все* подобные сделанные приостановки. Если выдается COMMIT, и есть нарушения данными хотя бы одного любого ограничения, Oracle молча подменит COMMIT на ROLLBACK. Для программиста это важное обстоятельство, так как произойдет отказ от *всех* изменений в пределах последней транзакции, без разбору. Если же выдается SET CONSTRAINT ... IMMEDIATE, транзакция не закроется, а программа получит в ответ сообщение об ошибке.

Упражнение. Испытайте действие отложенных проверок объявленных ограничений целостности на примере имеющегося правила внешнего ключа.

Вместо приостановки/возобновления проверки нескольких ограничений по отдельности можно выполнить эти действия для всех ограничений зараз, например:

```
SET CONSTRAINT ALL DEFERRED;
```

Если вместо слова ALL перечислить через запятую имена ограничений, перевод (в IMMEDIATE или DEFERRED) произойдет для ограничений из списка. Перевести все имеющиеся ограничения в нужное состояние можно еще командой ALTER SESSION, впрочем не добавляющей ничего нового к действию SET CONSTRAINT; например:

```
ALTER SESSION SET CONSTRAINTS = IMMEDIATE;
```

Ограничения, объявленные как DEFERRABLE, можно дополнить правилом проверки по умолчанию, например:

```
CREATE TABLE x (  
    a NUMBER CHECK ( a > 0 ) DEFERRABLE INITIALLY DEFERRED,  
    b NUMBER CHECK ( b > 1 ) DEFERRABLE INITIALLY IMMEDIATE );
```

Ограничение для столбца А нормально не проверяется (умолчательно приостановлено) и временно допускает вступление в силу. Ограничение для столбца В нормально проверяется и временно допускает приостановку действия. Само указание слов INITIALLY IMMEDIATE является умолчательным.

Упражнение. Проверить срабатывание указанных для столбцов таблицы X условий на заносимые значения.

Приостановка проверки ограничений может использоваться:

- ради удобства внесения некоторых видов изменений (например, при замене в таблице значения первичного ключа или же уникальной группы при наличии ссылок внешними ключами, как компенсация за отсутствие в Oracle определения UPDATE CASCADE; например, при наличии встречных ссылок внешними ключами при имитации связи «один-к-одному»);
- по необходимости (например, при ссылке внешним ключом на столбцы собственной таблицы; например, при наложении ограничений целостности на автоматически обновляемые materialized views).

Не все специалисты в реляционном подходе к проектированию БД разделяют мнение о целесообразности приостановки проверки ограничений, допускаемой в SQL, указывая на другие способы решения возникающих проблем. Нельзя не заметить, что во время приостановки действия ограничений БД может давать на запросы ответы, неправильные с точки зрения заложенной в нее модели предметной области. В любом случае подобная приостановка должна использоваться программистом исключительно как временная техническая мера.

8.4. «Долговременное» отключение ограничений целостности

Действующее в схеме объявляемое ограничение целостности можно отключить также на период, более долгий, чем транзакция, вообще безотносительно к последующим транзакциям:

```
ALTER TABLE dept MODIFY CONSTRAINT u_names DISABLE;
```

Мотивами для «долговременного» отключения проверки ограничений могут, например, стать:

- погоня за скоростью загрузки данных, когда заранее известно, что данные удовлетворяют всем ограничениям;
- сохранение доступности БД при поступлении отдельных нарушающих ограничения данных (например, оказалось, что паспорта с одним номером выданы двум разным лицам).

Допускается задавать отключенность ограничения изначально при его создании.

Включение ограничения аналогично отключению, но с указанием слова ENABLE. Как и при включении проверки после ее приостановки, здесь тоже возможны конфликты с данными, однако способы разрешения конфликтов — свои собственные.

8.4.1. Технология включения и выключения объявленных ограничений целостности

Если нормальный перевод ограничения в состояние ENABLED невозможен из-за накопленных противоречащих ограничению данных, Oracle предлагает два практических выхода из подобной ситуации:

- включить ограничение без проверки накопленных данных;
- выявление записей, нарушающих ограничение.

Оба решения представляют первоочередную ценность для больших таблиц.

8.4.1.1. Включение ограничения без выполнения проверки имеющихся данных

В некоторых случаях можно включить ограничение, отказавшись от проверки накопленных данных, и тогда оно будет проверяться только при внесении новых изменений в таблицу. При этом в таблице, возможно, сохранятся от прежних времен записи, нарушающие ограничение:

```
CREATE TABLE t (  
  c VARCHAR2 ( 1 ) CONSTRAINT ctest CHECK ( c IN ( 'a', 'b' ) ) DISABLE  
);
```



```
INSERT INTO t VALUES ( 'd' );
```

```
ALTER TABLE t MODIFY CONSTRAINT ctest ENABLE;  
-- Ошибка !
```

```
ALTER TABLE t MODIFY CONSTRAINT ctest ENABLE NOVALIDATE;
```

```
INSERT INTO t VALUES ( 'd' ) ;  
-- Ошибка !
```

Аналогично можно указывать NOVALIDATE или VALIDATE и при отключении проверки ограничения:

```
ALTER TABLE t MODIFY CONSTRAINT ctest DISABLE VALIDATE;  
-- Ошибка !
```

```
ALTER TABLE t MODIFY CONSTRAINT ctest DISABLE NOVALIDATE;
```

8.4.1.2. Выявление записей, нарушающих ограничение

Другая возможность состоит в том, чтобы при наличии строк, нарушающих ограничение, получить в распоряжение список физических адресов таких строк. Для этого:

а) нужно иметь специальную таблицу для хранения адресов строк. Типовой сценарий создания такой таблицы есть в `%ORACLE_HOME%\rdbms\admin\utlexcpt.sql` (для обычных таблиц) и в `utlexpt1.sql` (для индексно организованных таблиц). Имя таблицы, которую заводит сценарий `utlexcpt.sql`, — EXCEPTIONS. Подобных таблиц можно завести несколько;

б) выдать команду типа:

```
ALTER TABLE t MODIFY CONSTRAINT ctest ENABLE EXCEPTIONS INTO exceptions;
```

Если нарушения ограничения CTEST данными обнаружатся, в программу вернется ошибка, но при этом в указанную в команде таблицу EXCEPTIONS СУБД занесет список физических адресов строк, нарушающих ограничение. После корректировки данных в таблице T соответствующие ей строки в EXCEPTIONS нужно не забыть удалить самостоятельно, так как Oracle, разумеется, этого не сделает.

8.5. Более сложные правила целостности

Объявленные ограничения целостности в Oracle приспособлены для записи важных, но простейших видов дополнительных ограничений на помещаемые в базу значения данных. Более сложные правила (по современной терминологии «бизнес»-правила) могут быть заданы с использованием:

- триггерных процедур (выполняются СУБД Oracle);
- аппарата транзакций и блокировок (выполняется в СУБД Oracle);
- логики прикладной программы (воплощается клиентским приложением).

Порядок в указанном перечне соответствует убыванию гарантированности соблюдения правил целостности в фактически оказавшихся в базе данных (соответственно — увеличению риска получить в БД некорректные с прикладной точки зрения данные) по отношению к объявленным в схеме ограничениям.

9. Представления данных, или же виртуальные таблицы (views)

Буквальным переводом английского термина *view* в базах данных является «вид» [на данные]. Термин является сокращением от бытовавшего когда-то более точного названия *data view*. Содержательно правильным переводом могут быть термины «виртуальная», «выводимая», «производная» или же «синтезированная» таблица. Трудно удержаться от перевода «думная таблица», но русская терминология в области информационных технологий до подобного не созрела. Буквальный перевод в русском языке не прижился, а содержательно более ему предпочтительные оказались чересчур громоздкими. По этим и другим причинам у нас широко распространился «перевод» *представление* (данных). Все используемые и не вошедшие в оборот русские термины чем-то неудачны. Справедливости ради можно отметить, что и выбор оригинального термина не бесспорен. Предлагается, например, вместо *view* использовать «более правильный» термин *derived table*.

Название «виртуальная таблица», принятое здесь наравне с «представлением», переключается с названием «виртуальный столбец», официально появившимся в Oracle версии 11.

Понятие *view* попало в SQL из реляционной теории, но в выхолощенном виде, потеряв многие интересные качества. Там радикального отличия «основных отношений» от «производных» нет, и по сути все отношения являются как бы «представлениями» («видимостями») данных, допускающими внесение через себя изменений в БД, если только это теоретически позволительно.

В SQL термин используется для обозначения запроса SELECT, текст и разобранная структура которого хранятся в БД под определенным именем. Формальное основание такому хранению дает то обстоятельство, что любой запрос в SQL возвращает набор данных, структурированный в виде столбцов и строк, равно как в таблице, извлеченной из БД. Ничто не мешает взять этот набор данных в качестве источника для другого запроса. Если в запросе указать в качестве источника данных не реальную таблицу, а виртуальную («представление»), Oracle обнаружит подмену, вычислит сход запрос для нее и полученный результат предъявит для обработки основному запросу. Стоит только добавить, что это никогда не нарушаемая *логика* обработки, в то время как технически Oracle вполне может поступать иначе. Например, при определенных обстоятельствах Oracle может «растворить» текст *view* в основном запросе и вычислять основной запрос сразу и без предварительной фазы вычисления подзапроса-*view* или же вести вычисление и основного запроса, и данных подзапроса-*view* параллельно и так далее.

Таким образом для запросов SELECT, поступающих из приложения, нет никакой содержательной разницы, скрывается ли под именем источника данных виртуальная или реальная таблица, и тем самым выдержана преемственность реляционной модели. Разница может касаться только затрат ресурсов СУБД на выполнение.

Просто *таблицы* можно в противовес *view* уточнять словами «реальные», «основные», «базовые» (что то же самое), «исходные». Часто возникает соблазн назвать их «хранимыми», противопоставляя «вычислимым» *view*, но Oracle (и не только) дает пример «исходных» таблиц, не хранимых в БД: это так называемые X\$-таблицы, служащие администрированию.

Виртуальные таблицы позволяют решать в базе данных по крайней мере три важные задачи:

- удобства предоставления данных пользователю и упрощения вида запросов,
- ограничения доступа к данным, и
- приспособления к изменениям в модели предметной области путем учета этих изменений без перекройки накопленных в БД данных.

9.1. Примеры определений представлений данных

Ниже приводятся некоторые примеры создания и удаления виртуальных таблиц. Таблицы EMP и DEPT являются «реальными» (иначе — основными, или базовыми), а таблицы JOBS, COMMISSIONERS и EMPLOYEES — виртуальными, то есть представлениями данных.

```
CREATE VIEW jobs AS SELECT DISTINCT job FROM emp;
```

```
CREATE VIEW commissioners  
AS  
    SELECT *  
    FROM emp  
    WHERE comm IS NOT NULL  
;
```

```
CREATE VIEW employees ( name, place )  
AS  
    SELECT ename, loc  
    FROM emp LEFT JOIN dept  
    USING ( deptno )  
;
```

```
DROP VIEW employees;
```

9.2. Обновление БД через представления данных

Техника представлений позволяет дать программисту удобный взгляд на данные. Так как с точки зрения выборки данных эти виртуальные таблицы ничем не отличаются от реальных, со временем неизбежно возникает желание применять к ним не только SELECT, но и операторы DML. Смысл применения к представлению данных операторов INSERT, UPDATE и DELETE логично понимать как попытку внести изменения в основные таблицы, таким образом, чтобы создалось впечатление изменения таблицы виртуальной. То есть, строго говоря, речь идет не об «обновлении представлений данных» операторами DML, а об «обновлении БД» этими операторами «через представления».

Осуществить такие изменения в БД в каждом конкретном случае удастся не всегда, и по разным причинам: отчасти по объективным, отчасти по субъективным.

Oracle делит все представления данных на две категории: обновляемые (указанным выше образом) и необновляемые. Простейшим случаем обновляемого является представление, построенное запросом к единственной основной таблице. С версии 8 иногда стало возможно обновление представлений, построенных на основе запроса к более чем одной основной таблице. (При этом формальная обновляемость не гарантирует фактическое выполнение конкретной операции DML применительно к view: иногда оно может вступить в противоречие ограничениям целостности, связанным с таблицей. Примером того, как можно попытаться избежать подобной неопределенности, является включение в запрос для view поля первичного ключа — разумеется, когда таковой у основной таблицы имеется.)

В любом случае Oracle запрещает обновление данных, если в определении представления предложение SELECT содержит обобщение в том или ином виде, как то:

- DISTINCT
- агрегатные (обобщающие) и аналитические функции
- GROUP BY
- множественные операции

Если представление причислено к обновляемым, некоторые его столбцы могут оказаться закрытыми для обновления. Так, чтобы столбец допускал изменения, требуется, чтобы во фразе SELECT подлежащего запроса он:

- не претерпевал преобразований в виде выражений (функций, скалярных подзапросов и так далее);
- не был виртуальным столбцом или «псевдостолбцом»;
- если подлежащий запрос — соединение, операция DML должна прилагаться к унаследованному в определении подлежащего предложения SELECT первичному или уникальному ключу.

В любом случае через представление позволено изменять данные не более чем одной основной таблицы.

Более точный список правил предъявления операции DML к представлениям данных приведен в общем виде в документации по Oracle, а применительно к конкретным представлениям список обновляемых столбцов можно узнать из таблицы USER_UPDATABLE_COLUMNS словаря-справочника.

Из приводившихся выше определений:

- COMMISSIONERS допускает употребление себя в операторах DML без ограничений;
- JOBS не допускает употребление в операторах DML;
- EMPLOYEES допускает употребление в операторах DML, подразумевающих действительным объектом изменения основную таблицу EMP («изменить» столбец PLACE, например, будет нельзя).

Вот как в этом убедиться:

```
SQL> BREAK ON table_name
SQL> SELECT table_name, column_name, updatable, insertable, deletable
2 FROM user_updatable_columns
3 WHERE table_name IN ( 'JOBS', 'EMPLOYEES', 'COMMISSIONERS' )
4 ORDER BY 1
5 ;
```

TABLE_NAME	COLUMN_NAME	UPD	INS	DEL
COMMISSIONERS	EMPNO	YES	YES	YES
	ENAME	YES	YES	YES
	JOB	YES	YES	YES
	MGR	YES	YES	YES
	HIREDATE	YES	YES	YES
	SAL	YES	YES	YES
	COMM	YES	YES	YES
	DEPTNO	YES	YES	YES
EMPLOYEES	NAME	YES	YES	YES
	PLACE	NO	NO	NO
JOBS	JOB	NO	NO	NO

Радикально решить все проблемы обновления данных через представления способна триггерная процедура типа INSTEAD OF, с помощью которой разработчик получает шанс самостоятельно запрограммировать необходимые изменения в БД, отражающие, по его мнению, смысл применения INSERT, UPDATE или DELETE к любому представлению.

9.3. Ограничения целостности для представлений данных

Попытки внести в БД изменения через представления данных можно связать ограничениями. К определению представлений не применимы ограничения целостности, существующие для обычных таблиц (хотя в реляционном подходе виртуальные, производные отношения и могут иметь подобные ограничения), однако имеются два вида собственных: запрет непосредственных обновлений и ограничение возможных изменений областью видимости через view. Оба применяются к таблице целиком, а не к отдельным строке или столбцу.

9.3.1. Запрет непосредственных обновлений

При создании представления можно запретить изменение данных БД через него непосредственно командами DML:

```
CREATE VIEW employees ( name, location )
AS
SELECT ename, loc
FROM emp LEFT JOIN dept
USING ( deptno )
WITH READ ONLY
```

;

В таких представлениях «изменять» данные можно только опосредовано, путем изменения значений в подлежащих реальных таблицах. Такие представления могут служить жестким средством ограничения доступа к данным.

Для ограничения READ ONLY у виртуальных таблиц не предусмотрено собственное именование конструкцией CONSTRAINT, и оно всегда получает системное название в перечне в USER_CONSTRAINTS.

9.3.2. Сужение возможности непосредственных обновлений

Указание WITH CHECK OPTION при определении представления не запрещает применять к нему операции INSERT, UPDATE и DELETE вообще, а запрещает только те изменения, которые совершатся в подлежащих таблицах, но не будут обнаруживать себя в данных самого представления. В то же время в отсутствие такого указания ненаблюдаемые через представление изменения в БД будут допускаться. Создадим представление о сотрудниках отдела 10:

```
CREATE VIEW emp10
AS
    SELECT *
    FROM   emp
    WHERE  deptno = 10
;
```

Получаем:

```
SQL> INSERT INTO emp10 ( empno, deptno ) VALUES ( 1111, 20 );
```

1 row created.

```
SQL> SELECT empno, deptno FROM emp10 WHERE empno = 1111;
```

no rows selected

```
SQL> SELECT empno, deptno FROM emp WHERE empno = 1111;
```

EMPNO	DEPTNO
1111	20

```
SQL> ROLLBACK;
```

Rollback complete.

То есть, предъявив INSERT к представлению EMP10, мы добавили в таблицу EMP сотрудника, которого само представление нам не показывает.

Изменим определение EMP10:

```
CREATE OR REPLACE VIEW emp10
AS
    SELECT *
    FROM   emp
    WHERE  deptno = 10
WITH CHECK OPTION
CONSTRAINT only_dept10
;
```

Получаем:

```
SQL> INSERT INTO emp10 ( empno, deptno ) VALUES ( 1111, 20 );
INSERT INTO emp10 ( empno, deptno ) VALUES ( 1111, 20 )
*
```

ERROR at line 1:

ORA-01402: view WITH CHECK OPTION where-clause violation

```
SQL> INSERT INTO emp10 ( empno, deptno ) VALUES ( 1111, 10 );
```

1 row created.

```
SQL> ROLLBACK;
```

Rollback complete.

Для ограничения CHECK OPTION у представлений предусмотрено собственное именование конструкцией CONSTRAINT, что выше и сделано. Без этого ограничение получит в USER_CONSTRAINTS системное название.

9.4. Материализованные (овеществленные) представления данных

Материализованные представления данных («овеществленные воображаемые таблицы» — materialized views) появились в версии Oracle 8.1 как вариация одноименной категории объектов БД, предлагаемой стандартом SQL. Аналогично обычным представлениям, они предполагают хранение формулировки запроса SELECT к таблицам-источникам, однако вдобавок сохраняют и сам результат запроса в виде хранимой таблицы. Делается это с основной целью обеспечить более быстрый, или же попросту надежный, доступ к данным, ввиду отсутствия необходимости вычислять подзапрос по ходу обработки основного запроса и возможности вместо этого взять уже посчитанный результат. Обратной стороной является усложнение техники работы с данными, неизбежное вследствие раздвоения, возникающего на логическом уровне схемы (источники подзапроса — посчитанный результат). Так, для этого рода объектов следует предусмотреть и обеспечить синхронизацию данных.

Двумя основными областями использования материализованных представлений данных являются следующие:

- Локальное хранение данных, отобранных из таблиц в других базах в сети. По старой терминологии (до версии 8.1) такие локальные выжимки удаленных данных назывались snapshots, то есть «снимки». В целях обратной совместимости слово snapshot продолжает существовать в некоторых командах и названиях в Oracle до сих пор, наряду с materialized view; по сути эти два термина сегодня в Oracle синонимичны.
- Устройство вспомогательных таблиц-«спутников» у особенно больших таблиц, ради хранения подсчитанных заранее обобщений (GROUP BY и агрегатные значения). Такое применение характерно для БД, спроектированных по типу «склада данных» (data warehouse) в системах поддержки принятия решений и анализа данных.

Материализованные представления создаются, переопределяются и удаляются командами {CREATE | ALTER | DROP} **MATERIALIZED VIEW** ..., например:

```
CREATE MATERIALIZED VIEW jobsal
AS
  SELECT job, SUM ( sal ) AS sum_sal
  FROM    emp
  GROUP  BY job
;
```

В отличие от обычной таблицы, которую можно было бы построить на основе того же запроса, для хранимой таблицы в JOBSAL предусмотрена возможность автоматического обновления по результатам изменений в таблице EMP. Она обеспечена определенными правилами и соответствующими синтаксическими конструкциями, здесь не использованными. В жизни определение материализованных представлений данных обязательно будет так или иначе сопровождаться дополнительными уточнениями.

Реляционная теория предпочитает различать понятия snapshot и materialized view. Более того, термин materialized view она считает некорректным (и даже логически противоречивым), полагая правильным употребление на его месте snapshot, ввиду того, что именно в названии snapshot заложено возможное отставание «материализованных» данных от текущего состояния таблиц-источников.

9.5. Особенности именованных представлений данных

Именованные представления данных, как простые, так и материализованные, обладают рядом общих полезных свойств. Они позволяют достичь логических (вне связи с эффективностью) выгод, уже перечислявшихся ранее.

Эффективность же их использования в БД определяется следующими общими факторами.

- Простые представления не предоставляют данные немедленно из базы. Получение данных в них требует вычислений и работы СУБД.
- Простые представления не предъявляют требований к месту в БД, связанных так или иначе с объемами представляемых данных; память в БД расходуется только на хранение в словаре-справочнике определений представлений.
- Материализованные представления данных позволяют ускорить обращение к данным в БД.
- Материализованные представления данных требуют в БД дополнительного места для хранения результатов запроса к базе.
- Материализованные представления несут с собой риски рассинхронизации и рассогласования данных и требуют в связи с этим определенных организационных хлопот.

9.6. Представления данных, встроенные в запрос

(Неименованные) представления данных, встроенные в запрос, не создаются как хранимые в БД объекты, а используются только в виде составных частей запросов SQL. Оригинальное название — inline view, то есть представления, «встроенные в текст» основного запроса.

Формально неименованное представление данных — это подзапрос, подставленный на место источника данных в SELECT, или в команды DML на изменение данных. Примеры применения во фразе FROM предложения SELECT приводились выше и относительно очевидны.

Пример неименованного представления данных в операторе изменения значений UPDATE:

```
UPDATE ( SELECT e.ename, d.dname
          FROM   emp e INNER JOIN dept d
                USING ( deptno )
          WHERE  d.dname = 'SALES'
          AND    e.comm IS NULL
        )
SET ename = INITCAP ( ename )
;
```

Обратите внимание, что вложенный SELECT («неименованное представление данных») делает выборку из двух таблиц. Следовательно этот оператор UPDATE не сводим к привычно применяемому к одной таблице, по крайней мере простым образом.

Подзапросы подобного рода объединяет с обычными представлениями то обстоятельство, что они не допускают автоматическую возможность «изменений» всех своих столбцов. Такие «изменения» иногда делать можно (пример выше), а иногда нельзя. Правила разрешения правки значений как раз совпадают с существующими для обычных представлений. Если программисту их потребуется в конкретных обстоятельствах уточнить, то достаточно создать на основе подзапроса обычное представление и справиться о возможности правки полей в USER_UPDATABLE_COLUMNS. Однако если такой

подзапрос эпизодический и не требует сохранения в БД, Oracle позволяет попросту вставить его в оператор DML указанным выше способом.

Упражнение. Верните именам сотрудников отдела продаж, не получающих комиссионные, первоначальное написание заглавными буквами. Замените «SET ename = INITCAP (ename)» на «SET dname = INITCAP (dname)» и попробуйте повторить запрос. Замените порядок указания таблиц во фразе FROM и вновь повторите запрос.

Еще примеры:

```
DELETE
FROM ( SELECT e.empno, deptno
      FROM emp e INNER JOIN dept d
        USING ( deptno )
      WHERE e.job <> 'CLERK'
    )
WHERE deptno = 10
;

INSERT
INTO ( SELECT e.empno, deptno
      FROM emp e INNER JOIN dept d
        USING ( deptno )
      WHERE e.job NOT IN ( 'CLERK', 'SALESMAN', 'MANAGER' )
      WITH CHECK OPTION
    )
VALUES ( 1111, 10 )
;
```

Упражнение. В последнем операторе замените VALUES (1111, 10) на VALUES (1112, 30) и повторите запрос. Следом уберите указание WITH CHECK OPTION и снова повторите. Объясните результаты.

Вернем значения EMP:

```
ROLLBACK;
```


10. Объектные типы данных в Oracle

Помимо сравнительно простых встроенных типов данных — как взятых из стандартов SQL, так и собственных, — Oracle допускает использование составных. Это конструируемые типы объектов, рассчитанные на хранение в БД данных, имеющих внутреннюю структуру. Эта структура известна СУБД, и СУБД позволяет с ней работать. Объектные типы позволяют хранить и обрабатывать средствами СУБД «сложно устроенные данные» более продвинутом образом, нежели это позволяет техника «больших неструктурированных объектов» типов LOB. Ввиду наличия вполне определенного типа (даже если это тип коллекции), единичное объектное значение можно полагать за скаляр, хотя оно и не будет атомарным.

Хранение в столбцах таблицы значений в виде объектов, в смысле объектного подхода (ОП в программировании и моделировании), фирма Oracle впервые обеспечила в рамках так называемой «объектно-реляционной модели» начиная с версии Oracle 8. Некоторые существенные пробелы первой реализации (например, отсутствие наследования типов) были устранены в версии 9. Примеры ниже не выходят за рамки возможностей версии 9.2, позже которой, впрочем, никаких существенных нововведений по объектной части не наблюдалось. Объектные возможности Oracle в общем следуют определениям SQL:1999, однако делают это не пунктуально.

10.1. Программируемые типы данных и объекты в БД

10.1.1. Простой пример

Ниже приводится простой пример использования программируемых (объектных) типов.

Вначале требуется создать «тип», как разновидности хранимых элементов БД. Пример создания типа объекта (в SQL*Plus):

```
CREATE TYPE address_type AS OBJECT (  
    zip          CHAR      ( 6 )  
    , location   VARCHAR2  ( 200 )  
)  
/
```

Здесь типу ADDRESS_TYPE приписаны два «свойства» (по объектной терминологии), ZIP и LOCATION. В действительности для представления адреса в типе наверняка будет указано большее количество свойств, однако в ознакомительном примере их более пространный перечень излишен, и не добавит понимания техники.

Определение типа напоминает определение таблицы, однако в отличие от таблицы (а также стандарта SQL и от реляционного подхода) тип объекта в Oracle не имеет права содержать ограничений целостности (которые в таком случае можно было бы назвать «ограничениями целостности типа»). Если необходимо их указать, сделать это придется только по месту употребления типа, то есть в описании таблицы. Отсутствие возможности задавать ограничения целостности в определении типа расценивается некоторыми специалистами отрицательно.

На «логическом уровне» описания типов тип, определенный на основе одного-единственного свойства встроенного типа, стандарт SQL называет «особым» (distinct). Его использование позволяет (а) создавать в конечном счете на «физическом уровне» контролируемо однотипные столбцы (например, для связи «внешний ключ»), и (б) проводить (правда, на уровне общего описания) разницу между содержательно разными типами, вынужденно реализованными единообразно (например, «количество апельсинов» и «количество яблок»). Этого же можно достичь путем создания типов по следующей схеме:

```
CREATE TYPE oranges_type AS OBJECT ( pieces NUMBER ( 4 ) )  
/  
CREATE TYPE apples_type AS OBJECT ( pieces NUMBER ( 4 ) )  
/
```

Это более сильное решение, нежели даваемое «особыми» типами, поскольку различие типов при нем передается с логического уровня вплоть до схемы БД. Однако оно сделает невозможным привычное выполнение арифметических операций с «апельсинами» и с «яблоками», разрешив с ними работать только через методы (что теоретически правильно, но в работе неудобно).

Oracle поддерживает термин `distinct` для типов, создаваемых на уровне логического описания типов (Data Types Model) в своей системе Data Modeller (включенной в состав SQL Developer). Логические же описания типов, переводимых в «физической модели» в типы объектов БД, Oracle называет «структурными».

Понятие «особых типов», пусть и на логическом уровне, одобряется не всеми специалистами.

В соответствии с традициями объектного подхода (уместно вспомнить, что «объектной теории», в отличие от реляционной, не создано) Oracle разрешает использовать тип для создания «буквальных значений» объектного вида, по-другому — «объектных величин», и собственно объектов. Далее приводится сначала несколько примеров первого, а затем — второго.

10.1.2. Таблицы с объектными столбцами

«Буквальные значения» фактически позволяют работать со значениями, обладающими известной СУБД структурой и однозначно определяются набором значений элементов своей структуры.

Примеры использования типа `ADDRESS_TYPE` для определения столбца в обычной таблице:

```
CREATE TABLE odept1 (
  dname    VARCHAR2 ( 20 )
, deptno   NUMBER    ( 2 ) CONSTRAINT pk_odept1 PRIMARY KEY
, addr     address_type
);

CREATE TABLE oempl (
  ename    VARCHAR2 ( 20 )
, empno    NUMBER    ( 4 ) CONSTRAINT pk_oempl PRIMARY KEY
, deptno   NUMBER    ( 2 ) CONSTRAINT fk_oempl REFERENCES dept
, home     address_type
);
```

Столбцы `ADDR` и `HOME` можно с некоторой вольностью назвать «объектными атрибутами». Они *не* позволяют хранить объектные значения в виде самостоятельной сущности и ссылаться на них ссылками. Получить из БД такие значения можно только по обычным правилам поиска данных в таблице.

В выражениях явно указанные объектные значения формулируются с помощью конструктора. В отличие от других объектных систем, например, от Java, в Oracle конструктор умолчательно имеет список параметров, соответствующих свойствам типа. Примеры применения в операциях добавления данных:

```
INSERT INTO odept1
  ( deptno, dname, addr )
VALUES
  ( 10, 'RESEARCH', address_type ( '123456', 'Archangelsk' ) )
;

INSERT INTO oempl
  ( empno, ename, deptno, home )
VALUES
  ( 1111, 'SMITH', 10, address_type ( '789012', 'Samara' ) )
;
```

Oracle допускает определенные синтаксические вольности в записи выражения над объектными данными. Здесь и далее используются частные случаи возможных формулировок.

Пример применения в запросе о сотрудниках, «работающих по месту жительства», за исключением конкретного указанного адреса:

```

SELECT e.ename, d.dname
FROM   oempl e INNER JOIN odept1 d
      ON e.home = d.addr
WHERE  e.home <> address_type ( '345678', 'Leningrad' )
;

```

Пример показывает легкость формулирования сравнения составных величин, каковыми являются адреса. Сравнение осуществляется поэлементно, путем сравнением всех свойств по очереди. Увы, но простота формулировки не дает права программисту расслабляться и забывать об особых случаях сравнения с данными типа CHAR и с NULL. Так, присутствие NULL в буквальных объектных значениях запутывает проблему сравнения еще больше, чем для случая простых типов. Сравните:

```

SQL> SELECT 'OK' ok FROM dual
      2 WHERE address_type ( NULL, 'x' ) IS NOT NULL;

```

```

OK
--
OK

```

```

SQL> SELECT 'OK' ok FROM dual
      2 WHERE address_type ( NULL, 'x' ) = address_type ( NULL, 'x' );

```

no rows selected

То есть получается, что $x = x$ не дает TRUE, но притом x IS NOT NULL дает TRUE (x имеет значение).

В выражениях можно обращаться к буквальному объектному значению как к целому, а можно и к его отдельным свойствам. Во втором случае, как правило, требуется прибегать к псевдониму:

```

COLUMN home FORMAT A35
SELECT e.ename, e.home, e.home.zip FROM oempl e
;

```

10.1.3. Таблицы объектов

Созданный в БД тип, в соответствии со стандартом SQL, можно употребить и для создания «таблиц объектов»:

```

CREATE TABLE addresses1 OF address_type;

```

```

CREATE TABLE addresses2 OF address_type;

```

Хотя для этой категории хранимых элементов используется термин «таблица», такая таблица всегда содержит ровно один столбец, и именно объектного типа.

Занесение «строки» в такую таблицу может быть, в частности, оформлено так:

```

INSERT INTO addresses1 VALUES ( '123456', 'Archangelsk' );

```

Однако это определенная вольность, хотя и широко распространенная, а теоретически более правильна другая формулировка, с использованием конструктора:

```

INSERT INTO addresses1 VALUES ( address_type ( '123456', 'Archangelsk' ) );

```

Или даже так:

```

INSERT INTO addresses1 VALUES (
                                NEW address_type ( '123456', 'Archangelsk' )
);

```

По своему действию все три формулировки INSERT равносильны.

Пример запроса:

```
SELECT a.*, UPPER ( location ) FROM addresses1 a;
```

Объекты в таких таблицах хранятся как самостоятельные сущности, у которых имеется автоматически порождаемый СУБД внутренний уникальный идентификатор object ID, в соответствии с классическим объектным подходом позволяющий ссылаться на конкретные объекты из других таблиц или из программы. Сравнение элементов-«строк» в таблице объектов друг с другом происходит уже не по значениям свойств, как в случае объектного столбца в обычной таблице, а по значению object ID. Перейти на сравнение значений свойств позволяет функция VALUE, например:

```
SELECT dname FROM odept1 d, addresses1 a WHERE d.addr = VALUE ( a );
```

Сделан запрос об отделах, расположенных по адресам из таблицы ADDRESS1.

Не исключено, что создатели функции VALUE обсуждали другое ее название — LITERAL_VALUE. По крайней мере оно точнее описывает совершаемое действие: создание значения со структурой из объекта. Буквальные значения сравниваются друг с другом по значениям их свойств, а объекты — по значениям object ID.

10.1.4. Ссылки на объекты

Благодаря наличию внутренних идентификаторов object ID у объектов из таблиц объектов, и следующего из этого праву на них ссылаться, локализовать такие объекты становится возможным не только с применением обычной техники SQL, но и навигацией по ссылкам.

Пример создания обычной таблицы со столбцом для ссылки на хранимый в таблице объектов (типа ADDRESS_TYPE) элемент-объект:

```
CREATE TABLE odept2 (
  dname      VARCHAR2 ( 50 )
, deptno    NUMBER CONSTRAINT pk_odept PRIMARY KEY
, addr      REF address_type SCOPE IS addresses1      -- столбец-ссылка
);
```

Здесь описание ссылки сужено возможностью адресоваться только к объектам из таблицы ADDRESSES1. Допускаются варианты описания ссылки: ее нацеленность на содержимое конкретной таблицы можно не применять или же, напротив, усилить до аналогии со ссылочной целостностью (в этом примере аналогия с правилом внешнего ключа неполная).

Пример заполнения поля ADDR значением-ссылкой:

```
INSERT INTO odept2
  ( dname, deptno, addr )
VALUES
  ( 'RESEARCH', 10, ( SELECT REF ( a )          -- функция получения ссылки
                      FROM   addresses1 a
                      WHERE  a.location = 'Archangelsk'
                    )
  );
```

Пример допустимых оформлений обращения в свойства объекта через ссылку:

```
COLUMN deref(d.addr) FORMAT A40

SELECT d.dname, Deref ( d.addr ), d.addr.zip FROM odept2 d
;
```

Здесь третий по порядку столбец сформулирован с учетом допускаемой синтаксической вольности, в отсутствии которой для выдачи свойства адреса (ZIP) приходилось бы непременно прибегать к функции Deref.

Навигация по объектам в БД с помощью ссылок, и в том числе извлечение объекта из БД, возможны не только в запросах SQL, но и в программе на PL/SQL. Однако само разрешение обращаться к данным БД по ссылке в обход SQL подвергается некоторыми специалистами критике.

10.1.5. Пример использования методов объектов

Более сложные конструкции в описании типа позволяют задавать методы объектов и типов. Пример задания метода в типе объекта:

```
CREATE TYPE employee_type AS OBJECT (  
    name      VARCHAR2 ( 50 )  
    , hiredate DATE  
    , home     REF address_type  
    ,  
    MEMBER FUNCTION days_at_company RETURN NUMBER  
)  
/
```

Когда методов много, их заголовки перечисляются по очереди через запятую, общим списком со свойствами.

В описании *типа* приводится только *заголовок* методов. Для описания *тела* метода необходимо создать *тело типа* (полная аналогия пары *пакет — тело пакета*, имеющейся в PL/SQL):

```
CREATE TYPE BODY employee_type AS  
MEMBER FUNCTION days_at_company RETURN NUMBER IS  
    BEGIN  
        RETURN TRUNC ( SYSDATE - hiredate );  
    END;  
END;  
/
```

Пример использования типа в создании таблицы:

```
CREATE TABLE sailors ( ship VARCHAR2 ( 30 ), emp employee_type );  
-- в этом контексте EMPLOYEE_TYPE — это имя типа
```

Использование конструктора типа для заполнения таблицы:

```
INSERT INTO sailors VALUES  
    ( 'Ninna', employee_type ( 'Frank Naude', SYSDATE, NULL ) )  
;  
-- в этом контексте EMPLOYEE_TYPE — это конструктор
```

Использование свойств и метода типа для запроса к таблице:

```
COLUMN emp FORMAT a60  
SELECT * FROM sailors;  
  
SELECT x.ship, x.emp.name, x.emp.days_at_company ( ) FROM sailors x;  
-- скобки после DAYS_AT_COMPANY сообщают, что это метод, а не свойство
```

10.2. Коллекции

Коллекции позволяют хранить в поле строки таблицы сразу множество значений: скалярных, объектов или ссылок на объект. Таким образом в БД они представляют собой еще один способ группировки элементов, дополнительно к объектным таблицам. Oracle относит коллекции к объектным возможностям своей СУБД, и поэтому допускаются многоуровневые коллекции.

Формально коллекции определяются через тип, и поэтому явного нарушения скалярности данных в таблицах Oracle коллекции не создают: в столбце таблицы по-прежнему хранятся значения определенного типа. Однако содержательно они все-таки моделируют хранение набора величин как целого, и в виду этого вызывают определенный скепсис со стороны знатоков реляционной модели.

Начиная с версии 8 в диалекте SQL Oracle имеется два вида коллекций: вложенные таблицы (неупорядоченное множество) и массивы VARRAY (упорядоченный список). Они соответствуют двум видам коллекций в стандарте SQL, но реализованы с некоторыми вольностями.

10.2.1. Вложенные таблицы

Вложенные таблицы (nested tables) в Oracle есть термин для обозначения возможности хранить в поле строки сразу множество значений («таблицу» значений). Обычную таблицу в Oracle, некоторые столбцы которой описаны как «вложенные таблицы», всегда можно препроектировать в информационно равносильный набор обычных таблиц с «единичными» столбцами.

Пример употребления вложенных таблиц:

```
CREATE TYPE colourset_typ AS TABLE OF VARCHAR2 ( 32 )
/

CREATE TABLE colour_models (
    model_type VARCHAR2 ( 12 )
    , colours    colourset_typ
)
NESTED TABLE colours STORE AS colour_model_colours_tab
-- фраза NESTED TABLE вынужденная, но может содержать указания оптимизации хранения
;

INSERT INTO colour_models VALUES (
    'RGB'
    , colourset_typ ( 'RED', 'GREEN', 'BLUE' )
);
-- в этом контексте COLOURSET_TYP — конструктор коллекции

COLUMN colours FORMAT A60
SELECT * FROM colour_models
;
```

Пример приоткрывает то обстоятельство, что технически перечень самих значений из хранимых в столбце наборов располагается не непосредственно в основной таблице, а в отдельной вспомогательной (служебной), и ей мы *обязаны* дать, по меньшей мере, имя. Однако помимо этого, во фразе NESTED TABLE при создании основной таблицы мы *имеем право* указать некоторые подробности организации доступа к этой служебной таблице и особенности хранения (например, табличное пространство).

10.2.2. Массивы VARRAY

По внутренней технической организации хранят списки в полях типов LOB по правилам, допускаемым для этих типов: как вместе со строкой таблицы (короткие списки), так и в отдельном сегменте LOB (длинные списки). В любом случае специальной вспомогательной таблицы для хранения значений массива здесь не требуется. Простой пример:

```
CREATE TYPE addresslist_typ IS VARRAY ( 5 ) OF address_type;
/
```

```
CREATE TABLE participants (
    name          VARCHAR2 ( 20 )
, locations addresslist_typ
);
```

При создании таблицы со столбцом-массивом VARRAY не требуется приводить дополнительных указаний (как для столбца вложенной таблицы), однако имеется возможность их применить при необходимости в том.

Добавление и выборка данных внешне не отличается от осуществляемых для вложенной таблицы, например:

```
INSERT INTO participants VALUES
( 'Einstein'
, addresslist_typ ( address_type ( '123456', 'Archangelsk' )
                    , address_type ( '789012', 'Samara' )
                    -- конструкторы простого объекта
                    )
-- конструктор массива VARRAY
);
```

10.2.3. Перевод данных в коллекции к табличному представлению и другие возможности

Хотя столбец-коллекция в таблице может показаться удобным для моделирования данных предметной области, работать с такими данными в SQL не обязательно просто. На помощь в этом приходит особая функция TABLE. Она придумана для «разворачивания» элементов коллекции в список строк, к которому можно уже привычным образом применять охватывающие запросы. Примеры:

```
SELECT *
FROM   TABLE ( SELECT colours
                FROM   colour_models
                WHERE  model_type = 'RGB'
                );

SELECT *
FROM   TABLE ( SELECT locations
                FROM   participants
                WHERE  name = 'Einstein'
                );
```

Упражнение. Проверить работу последних приведенных запросов.

Для разворачивания многоуровневых коллекций предусмотрен особый случай употребления функции TABLE в сочетании с соединением (join). Это довольно своеобразная конструкция, в которой:

- синтаксис соединения, взятый из стандарта SQL, не имеет силы (и, в частности, запись полуконечного соединения оформляется только посредством '(+)'), и
- порядок перечисления источников во фразе FROM строго определенный, от более общего слева к более подробному вправо.

Пример выдачи в табличной форме сведений об именах участников и адресах из таблицы PARTICIPANTS:

```
COLUMN location FORMAT A30
SELECT
```

```

    name
  , l.zip
  , l.location
FROM
    participants
  , TABLE ( locations ) l
;

```

Упражнение. Выдать таблицу имеющихся сочетаний <название цветовой модели, название компоненты> по данным из COLOUR_MODELS. Для ссылки на элемент (единственный) коллекции COLOURS из таблицы COLOUR_MODELS использовать псевдостолбец OBJECT_VALUE.

Сворачиванию значений столбца во вложенную таблицу помогает агрегирующая функция COLLECT:

```

SELECT  deptno, CAST ( COLLECT ( job ) AS colourset_typ ) jobs
FROM      emp
GROUP BY deptno
;

```

С версии 11.2 функция COLLECT позволяет строить коллекции с отсевом дубликатов и с определенным порядком элементов:

```

SELECT  deptno
        , CAST ( COLLECT ( DISTINCT job ) AS colourset_typ ) jobs
FROM      emp
GROUP BY deptno
;
SELECT  deptno
        , CAST ( COLLECT ( job ORDER BY sal ) AS colourset_typ ) jobs
FROM      emp
GROUP BY deptno
;

```

Упражнение. Выдать в SQL*Plus команду COLUMN jobs FORMAT A70 WORD и проверить работу последних приведенных запросов.

Приведенный перечень превращений множественных данных из одной формы в другую не исчерпывается сказанным и продолжаем. Сверх того, отдельные возможности программной обработки данных-коллекций предусмотрен в PL/SQL.

10.2.4. Различия в употреблении

Вложенные таблицы и массивы VARRAY различаются по содержательному употреблению и технике исполнения. Формальные свойства употребления вложенных таблиц и массивов VARRAY, хотя большей частью совпадают, имеют и естественные различия. Например, с версии 10 для определенного класса вложенных таблиц (элементы которых допускают сравнение друг с другом, в частности встроенных скалярных типов) возможны множественные операции. Еще пример — только для вложенных таблиц возможна проверка на пустоту:

```

SQL> SELECT 'ok' FROM dual WHERE colourset_typ ( ) IS EMPTY;

'O
--
ok

```

Вложенные таблицы, в отличие от массивов VARRAY, допускают внесение изменений техникой inline view («встроенное представление данных»), образованных разворачиванием таких таблиц в перечень строк применением функции TABLE. Пример:


```

INSERT INTO TABLE ( SELECT colours
                        FROM   colour_models
                        WHERE  model_type = 'RGB' )
VALUES ( 'BLACK' )
;

```

Упражнение. Удалить командой SQL DELETE компоненту BLACK из цветовой модели RGB.

10.3. Тип XMLTYPE

Этот встроенный объектный тип для работы в БД с документами XML появился в версии 9. До этого наиболее подходящим для хранения документов XML был тип CLOB. Тип XMLTYPE технически может либо по-прежнему базироваться на CLOB, либо иметь в БД структуру объекта (начиная с версии 9.2 и при использовании XML DB). Помимо пользовательской направленности, тип XMLTYPE активно применяется в последних версиях Oracle для внутренней организации БД.

СУБД и БД Oracle предлагают широкий спектр возможностей по использованию типа XMLTYPE в связи с документами XML. В целом они соответствуют описаниям, объединенным в стандартах SQL:2003+ общим названием SQL/XML (по-другому — SQLX; «XML-расширения языка SQL»). Ниже для знакомства приводятся только простые примеры.

10.3.1. Простой пример

Создание и пополнение таблицы, в которой решено хранить описания книг в формате XML — ради сохранения формата источника или же в силу отсутствия фиксированной структуры у описания книги:

```

CREATE TABLE books (
    id          NUMBER
, description XMLTYPE
);

INSERT INTO books VALUES (
    100
, xmltype (
    '<?xml version="1.0"?>
    <cover>
      <title>Java Programming with Oracle JDBC</title>
      <author>Donald Bales</author>
      <publisher>Oreilly and Associates</publisher>
      <pubdate>December 2001</pubdate>
      <isbn>0-596-00088-x</isbn>
      <pages>496</pages>
    </cover>'
    )
);
-- ... здесь Oracle разрешает вместо конструктора XMLTYPE ( '...' ) написать просто '...'

SET LONG 1000

SELECT id, description FROM books;

SELECT id, b.description.XMLDATA FROM books b;

```

XMLDATA — специально созданный для XMLTYPE «псевдостолбец». В данном примере его может заменить функция XMLSERIALIZE, одна из многих существующих для XMLTYPE.

Упражнение. Попробовать занести в поле DESCRIPTION неправильно оформленный документ XML и проследить реакцию СУБД.

Пример выборки с использованием условия отбора на языке XPath:

```
SELECT b.id, XMLSERIALIZE ( DOCUMENT b.description AS CLOB )
FROM   books b
WHERE  b.description.EXISTSNODE ( '/cover[author="Donald Bales"] ' ) =1;
```

10.3.2. Таблицы данных типа XMLTYPE

По аналогии с таблицами объектов, проектируемых самостоятельно, можно создавать таблицы документов XMLTYPE:

```
CREATE TABLE xbooks OF XMLTYPE;
```

Работать с ними можно, как и с прочими таблицами объектов:

```
INSERT INTO xbooks VALUES (
    xmltype (
        '<?xml version="1.0"?>
        <cover>
            <title>Java Programming with Oracle JDBC</title>
            <author>Donald Bales</author>
            <publisher>OReilly and Associates</publisher>
            <pubdate>December 2001</pubdate>
            <isbn>0-596-00088-x</isbn>
            <pages>496</pages>
        </cover>'
    )
);
```

-- ... здесь Oracle разрешает вместо XMLTYPE ('...') указать просто '...'

```
SELECT XMLSERIALIZE ( DOCUMENT OBJECT_VALUE AS CLOB ) FROM xbooks;
```

OBJECT_VALUE — псевдостолбец, позволяющий сослаться на логически единственный (объектный) столбец в таблице документов XML.

В этом примере данные XML будут храниться как CLOB. Более сложный пример — создание таблиц объектов типа XMLTYPE, где документы XML хранятся в виде таблицы объектов, а не как CLOB. Вот как это могло бы выглядеть в какой-нибудь БД:

```
CREATE TABLE oxbooks OF XMLTYPE
    XMLSCHEMA "http://www.oracle.com/xbooks.xsd"
    ELEMENT "Book"
;
```

Чтобы таблица OXBOOKS была таким образом заведена в действительности, требуется, чтобы схема XML <http://www.oracle.com/xbooks.xsd> была предварительно определена («зарегистрирована») в «репозитории» XML DB (сама XML DB со своим репозитарием автоматически включена в состав типовым образом созданой БД). Достоинство такого описания столбца XMLTYPE в том, что он позволяет хранить не произвольные документы XML, а только типизированные схемой XML. Тем самым зарегистрированная схема XML используется как средство ограничения целостности хранимых данных XML, налагаемое в таблице Oracle по правилам технологии XML.

10.3.3. Преобразование табличных данных в тип XMLTYPE

Ряд функций, воспроизведенных по описаниям SQL/XML, позволяет просто осуществлять преобразование обычных табличных данных в формат XML. В Oracle реализованы следующие функции из стандартного набора SQL/XML:

- XMLELEMENT
- XMLATTRIBUTES
- XMLAGG
- XMLCONCAT
- XMLFOREST
- XMLPI^{[10.2-)}
- XMLCOMMENT^{[10.2-)}
- XMLROOT^{[10.2-)}
- XMLSERIALIZE^{[10.2-)}
- XMLPARSE^{[10.2-)}

^{[10.2-)} Начиная с версии 10.2.

Вдобавок к этому Oracle SQL содержит ряд расширений по отношению к SQL/XML и несколько собственных функций для выполнения подобных преобразований:

- XMLCOLATTVAL
- SYS_XMLGEN (распространяется на строки)
- SYS_XMLAGG (распространяется на группы GROUP BY)
- XMLCDATA^{[10.2-)}
- XMLSEQUENCE^{(-11.1]} (только курсорный вариант)

^{[10.2-)} Начиная с версии 10.2.
^{(-11.1]} С версии 11.2 поддерживается ради обратной совместимости.

Следующие примеры поясняют действие некоторых из перечисленных функций:

```
SET LONG 2000
```

```
SELECT XMLELEMENT ( "employee", ename ) AS employee
FROM emp;
```

```
SELECT
  XMLELEMENT (
    "employee"
    , XMLATTRIBUTES ( ename AS "name", comm AS "commission" )
  ) AS employee
FROM emp;
```

```
SELECT
  XMLELEMENT (
    "employee"
    , XMLFOREST ( ename AS "name", comm AS "commission" )
  ) AS employee
FROM emp;
```

```
SELECT
  XMLELEMENT (
    "employee"
    , XMLCOLATTVAL ( ename AS "name", comm AS "commission" )
  ) AS employee
FROM emp;
```

```
SELECT
  XMLELEMENT (
    "department"
    , XMLATTRIBUTES ( deptno AS no )
  ) AS department
, XMLAGG ( XMLELEMENT ( "employee", ename ) ) AS employees
FROM emp
GROUP BY deptno
;
```

Обратите внимание, что в результатах выдаются столбцы типа XMLTYPE:

```
SELECT
  XMLELEMENT ( "name", ename ).EXISTSNODE ( '/name' )
FROM emp
;
```

Интересно, что функции для данных XML иногда позволяют решать в SQL задачи, не имеющие никакого к XML отношения. Примером может послужить выдача из БД требуемых значений списком, а не в виде строк таблицы. Конкретно она просто решается применением функции агрегирования LISTAGG, однако эта функция недоступна в версиях ранее 11.2, в которой она появилась, и ее результат ограничен размером строки в 4000 байт.

В следующем запросе функции и метод XMLTYPE используется как чисто техническое средство для достижения цели, никоим образом не связанной с этим типом:

```
WITH xview AS (
  SELECT XMLAGG ( XMLELEMENT ( "empno", empno || ', ' ) ) AS xempno
  FROM emp
  WHERE deptno = 10
)
SELECT
  RTRIM ( xv.xempno.EXTRACT ( '//text ( )' ), ', ' ) "Список через запятую"
FROM
  xview xv
;
```

Для удобства формулировки запроса подзапрос вынесен из основного текста.

Ответ:

```
Список через запятую
-----
7782, 7839, 7934
```

Если подобное решение кажется разработчику чересчур громоздким, он волен запрограммировать на PL/SQL собственную агрегирующую функцию, что позволяет делать Oracle, или же, палиативно, воспользоваться функцией GETXMLTYPE из состава встроенного пакета DBMS_XMLGEN.

10.4. Тип ANYDATA

Имеется, начиная с версии 9. Позволяет хранить в столбце таблицы данные одновременно разных типов. Используется с объектным типом ANYDATA, имеющим свои конструкторы и методы. Пример:

```
CREATE TABLE t ( x ANYDATA );

INSERT INTO t VALUES ( ANYDATA.CONVERTNUMBER ( 5 ) );
INSERT INTO t VALUES ( ANYDATA.CONVERTDATE ( SYSDATE ) );
INSERT INTO t VALUES ( ANYDATA.CONVERTVARCHAR2 ( 'hello world' ) );
INSERT INTO t VALUES ( ANYDATA.CONVERTOBJECT
                        ( address_type ( '789012', 'Murmansk' ) ) );
SELECT t.x.GETTYPENAME ( ) typename FROM t t;
```

Метода извлечения значения нефиксированного типа в SQL не предусмотрено, и извлечение приходится программировать в два захода: сначала узнать реальный тип значения, а потом уже применить соответствующий типу метод извлечения. Пример ниже подсказывает, как можно при желании составить функцию выборки данного, доступную в SQL, а не только в программе:

```
SET SERVEROUTPUT ON
DECLARE
  val VARCHAR2 ( 30 );
  anumber ANYDATA;
BEGIN
  SELECT t.x INTO anumber
  FROM t t
```

```

WHERE  t.x.GETTYPENAME ( ) = 'SYS.NUMBER'
-- в нашей таблице такая строка единственная
;
DBMS_OUTPUT.PUT_LINE ( 'Ret code: ' || anumber.GETNUMBER ( val ) );
DBMS_OUTPUT.PUT_LINE ( 'A number: ' || val );
END;
/

```

Методы типа ANYDATA описаны в документации по Oracle.

Тип ANYDATA, как и XMLTYPE, находит употребление во внутренней организации БД. В частности он используется в построении внутренних «автоматических очередей», лежащих в основе «поточков данных», используемых, в свою очередь, для автоматического переноса данных в Oracle.

11. Вспомогательные виды хранимых объектов

Основными инструментами моделирования предметной области в БД является в Oracle таблица данных и представление данных. Однако в любой настоящей БД под управлением Oracle всегда имеется определенное количество хранимых объектов других видов. Это вспомогательные, служебные объекты, призванные предоставить потребителю БД определенное удобство ее использования и определенную свободу действий. Основные категории таких объектов, формально хранимых в БД, но для описания предметной области бесполезных, описываются ниже.

11.1. Генератор последовательности из чисел

*В этом царстве люди нарождались и неведомо куда девались.
Сказки и легенды Пушкинских мест*

В первых описаниях реляционной модели было замечено, что требуемый для отношений в этой модели ключ не всегда легко задать. Тогда же предложили для решения проблемы заводить в отношениях «искусственный» (artificial) ключ, то есть служащий исключительно цели сообщения отношению необходимого правила ключа и не имеющий никакого содержательного (моделирующего предметную область) смысла. В жизни для этого обычно заводят ключ из одного атрибута числового типа, но так делается ради удобства: в отдельных случаях может оказаться целесообразным употребить несколько столбцов, равно как и прочие типы.

SQL не требует в таблицах первичного ключа (и вообще никакого), однако допускает его существование; для таблиц с первичным ключом сказанное переносится в SQL.

11.1.1. Создание и использование генератора

Когда разработчик решает завести в таблице искусственный первичный ключ, перед ним встает техническая задача: обеспечить заполнение столбцов ключа уникальными значениями. Иногда можно найти простой выход из положения, например, для одностолбцового ключа типа DATE можно брать значения из SYSDATE. Пригодность такого решения определяется конкретикой использования таблицы в приложении. Но оно не подойдет для всех таблиц и для числового столбца, наиболее популярного в роли первичного ключа. Некоторые типы СУБД для одностолбцового числового ключа вводят особый «автоинкрементный» тип данных. Oracle же, и отчасти IBM, предлагают брать в таких случаях значения из специального объекта хранения — датчика чисел (sequence; полное название — *sequence generator*, по-русски *породителя*, или *генератора, последовательности* чисел). Оба решения в разное время post factum попали в стандарт SQL, так что в Oracle автоинкрементный столбец оказался тоже включен, но с запаздыванием, с 12-й версии, причем с теми же свойствами, что у самостоятельного генератора.

Примеры создания генератора последовательности чисел и запросов к нему на выдачу очередного (NEXTVAL) и текущего (CURRVAL) значений:

```
CREATE SEQUENCE proj_numbers;

INSERT INTO proj ( projno, pname )
VALUES ( proj_numbers.NEXTVAL, 'DELTA' );

UPDATE proj SET projno = projnumbers.NEXTVAL WHERE projno = 16;

SELECT proj_numbers.CURRVAL FROM dual;
```

Замечания

- Порождаемые числа уникальны в рамках БД в целом, и отдельных сеансов, в частности.

- С точки зрения БД генератор последовательности — хранимый объект (подобно таблице) и может использоваться разными сеансами по мере надобности. По этой причине получаемая *отдельным* сеансом последовательность чисел не обязана быть плотной и может содержать разрывы.
- CURRVAL выдает значение в рамках сеанса, доступное только после предшествующей выдачи NEXTVAL. Фактически это последнее значение NEXTVAL, полученное в конкретном сеансе (но не вообще от генератора).
- Последовательность чисел порождается СУБД *безотносительно* к открытию и завершению транзакций.

Более сложный пример определения генератора:

```
CREATE SEQUENCE dept_numbers
MINVALUE 0 MAXVALUE 2000 -- минимальное и максимальное допустимые значения
START WITH 1000          -- первое выдаваемое число
INCREMENT BY -10         -- шаг изменения чисел в последовательности
CYCLE                    -- дойдя до границы, переключиться на противоположную
CACHE 20                 -- способ ускорить выдачу при особо частых обращениях
;
```

Свойство CYCLE способно привести через определенное время к повторениям значений и фактически отменит основное качество такого генератора. По умолчанию действует свойство NOCYCLE.

Удаление:

```
DROP SEQUENCE dept_numbers;
```

Использование генератора в выражениях SQL вовсе не обязательно требует дополнительного программирования. Примеры применения генератора чисел в множественных операциях DML:

```
CREATE SEQUENCE seq;

CREATE TABLE emps AS SELECT 0 id, ename FROM emp;

UPDATE emps SET id = seq.NEXTVAL;

CREATE TABLE empss AS SELECT seq.NEXTVAL id, ename FROM emp;
```

11.1.2. Изменение свойств генератора

Большую часть свойств генератора можно изменять командой ALTER SEQUENCE. Например:

```
ALTER SEQUENCE proj_numbers NOCACHE MINVALUE -1000 NOMAXVALUE;
```

Однако изменить таким же путем текущее значение нельзя. В то же время это иногда требуется, например, после импорта данных. В таких случаях можно сделать следующее.

Во-первых, можно удалить генератор и воссоздать его заново с требуемым текущим значением. При этом есть шанс ошибиться с правильным воспроизведением прочих свойств.

Во-вторых, в качестве искусственной меры можно временно изменить шаг приращения на подходящую величину, обратиться к NEXTVAL, добившись нужного текущего значения, и вернуть приращение назад.

В-третьих, пользователь SYS способен внести желаемую величину непосредственно в основную таблицу словаря-справочника SEQ\$. Это можно рекомендовать в последнюю очередь и с соблюдением определенных предосторожностей.

Существующие значения свойств генератора можно взять из таблицы словаря-справочника USER_SEQUENCES. Например, последнее значение можно получить так:

```
SELECT increment_by FROM user_sequences WHERE sequence_name = 'SEQ';
```

Вычтем эту величину из целевой; результат укажем в команде ALTER SEQUENCE seq INCREMENT BY ...; сделаем запрос к seq.NEXTVAL и вернем начальное значение приращения командой ALTER SEQUENCE.

Этот вариант замены текущего значения следует признать наиболее безопасным. Он легко автоматизируется, однако, применяя его, нужно обеспечить отсутствие обращений к генератору со стороны посторонних сеансов в промежутках между совершаемыми операциями.

11.1.3. Применение генератора чисел

Важный для разработчика схемы БД вопрос, сколько генераторов числовых последовательностей заводить? Умозрительно допустимы два крайних варианта: (а) по одному генератору на каждую таблицу с искусственным первичным ключом, и (б) один на все таблицы разом. Издержки первого варианта — это хлопоты создания и поддержки, а второго — возможные задержки получения сеансами чисел в силу общей высокой частоты обращений.

Однако в жизни искусственный ключ часто оказывается таковым лишь отчасти. В схеме SCOTT, например, он может использоваться для порождения табельных номеров сотрудников (EMP.EMPNO) и номеров отделов (DEPT.DEPTNO). В обоих случаях действуют правила максимально допустимого количества десятичных цифр (4 и 2) и положительности значения. Это довод в пользу скорее индивидуальных генераторов для таблиц (хотя и не обязательно доводить дело до крайности), так как придется создавать генераторы последовательностей со вполне определенными свойствами, по сути дополняющими определение типов столбцов:

```
CREATE SEQUENCE emp_empno_seq MINVALUE 1 MAXVALUE 9999
;
CREATE SEQUENCE dept_deptno_seq MINVALUE 1 MAXVALUE 99
;
```

К сожалению, формальной увязки генераторов со столбцами в текущих версиях Oracle по-прежнему нет, и с недоразумениями из-за попаданий в столбец непредусмотренных величин приходится бороться недостаточно надежными организационными мерами. Вопрос, стоит ли заводить ограничения целостности наподобие CHECK (empno > 0), или же, вместо этого, триггерные процедуры типа ON INSERT/UPDATE OF emp, разработчик БД решает сам, сообразуясь с обстоятельствами.

11.1.4. Встроенный генератор

С версии 12 разработчик получил возможность приписать столбцу генератор в определении таблицы. Это делается двояко. Во-первых, указанием в определении столбца характеристики GENERATED [ALWAYS] AS IDENTITY:

```
ALTER TABLE dept_copy ADD
  ( autonum NUMBER ( 10 ) GENERATED AS IDENTITY )
;
```

СУБД устанавливает значения в это столбец самостоятельно при INSERT:

```
INSERT INTO dept_copy ( dname, loc ) VALUES ( 'HQ', 'RENO' )
;
```

Проверка:

```
SQL> SELECT * FROM dept_copy WHERE dname = 'HQ';

DEPTNO DNAME          LOC          AUTONUM
```


-----	-----	-----
HQ	RENO	1

В результате такого объявления столбца Oracle создает генератор с системным именем и использует его автоматически при INSERT (прибегать к NEXTVAL не требуется), однако это обычный генератор. При желании, узнав его имя, к нему можно обращаться для получения чисел в любых других целях.

В таком определении столбца установить ему значение явочным порядком (INSERT, UPDATE) не удастся. Однако это разрешается при несколько ином определении:

```
INSERT INTO dept_copy ( autonum, dname ) VALUES ( 1000, 'PROD' )
-- Ошибка !
;
ALTER TABLE dept_copy MODIFY (
    autonum NUMBER GENERATED BY DEFAULT AS IDENTITY
                ( START WITH 100 INCREMENT BY 1000 )
);
INSERT INTO dept_copy ( autonum, dname ) VALUES ( 1000, 'PROD' )
-- OK!
;
INSERT INTO dept_copy ( dname ) VALUES ( 'PROD1' )
-- OK!
;
```

Дополнительно допускается формулировка **GENERATED BY DEFAULT AS IDENTITY ON NULL** (с очевидным смыслом).

11.2. Каталог операционной системы

Объект вида «каталог» («директория», directory; буквально — «выстроенный в линию», строй, ряд, от латинского dirigere; то есть «систематизированный» «список» файлов) используется для регулирования доступа СУБД к файлам в каталогах файловой системы ОС.

Пример создания (CREATE) или изменения (OR REPLACE):

```
CREATE OR REPLACE DIRECTORY extfiles_dir AS 'c:\crs';
```

Примеры употребления:

```
SELECT
    DBMS_LOB.GETLENGTH ( BFILENAME ( 'EXTFILES_DIR', 'sql.pdf' ) )
    AS "Bytes in the file:"
FROM dual
;

ALTER TABLE proj ADD ( description BFILE );

INSERT INTO proj ( projno, pname, description )
VALUES (
    3000
, 'YOTA'
, BFILENAME ( 'EXTFILES_DIR', 'sql.pdf' )
);

SELECT pname, DBMS_LOB.GETLENGTH ( description ) FROM proj;
```

Обратите внимание, что имя каталога EXTFILES_DIR в функцию BFILENAME передается строкою текста и не подвергается (в данном случае) автоматическому возвышению регистра в процессе обработки команды SQL. Такое определение функции BFILENAME следует признать удобным более для разработчиков, чем для пользователей Oracle, но оно не является исключением.

Объект вида «каталог» отличается от большинства объектов прочих видов тем, что является «внесхемным», наподобие некоторых других объектов, как например, создаваемых с уточнением PUBLIC (PUBLIC SYNONYM и др.). Технически это оформляется так: эти объекты всегда принадлежат пользователю SYS, кем бы они не создавались, причем создающий их фактически пользователь должен иметь системную привилегию CREATE ANY DIRECTORY (о привилегиях говорилось выше). Но, в отличие от объектов PUBLIC некоторых других категорий, объекты DIRECTORY не доступны пользователям БД автоматически, и их доступность регулируется объектными привилегиями READ и WRITE. Так, «пример употребления» выше проработает, если вместо команды CREATE ... extfiles_dir ... (как было) выдать:

```
CONNECT / AS SYSDBA
```

```
CREATE OR REPLACE DIRECTORY extfiles_dir AS 'c:\crs';
```

```
GRANT READ ON DIRECTORY extfiles_dir TO scott;
```

```
CONNECT scott/tiger
```

... и уже далее код «употребления».

11.3. Связь с другой БД

Связь с другой БД (database link) позволяет пользователю, подсоединенному к одной БД, обращаться к объектам из другой БД, организуя тем самым распределенные вычисления в базах данных.

Пример:

```
CREATE DATABASE LINK anotherdb
  CONNECT TO scott
  IDENTIFIED BY tiger
  USING 'oraserv:1521/orcl.class'
;
```

После слова USING, в кавычках указан сетевой адрес нужной БД, в данном случае образованный именем компьютера oraserv, номером порта 1521 и именем службы БД orcl.class. Чтобы выдать такую команду самостоятельно, пользователю необходимо иметь полномочие (привилегию) CREATE DATABASE LINK. (На практике администратор охотнее откажется наделять пользователя дополнительным полномочием, а вместо этого сам создаст ему связь с другой БД.)

Пример употребления:

```
SELECT ename, dname
FROM emp e, dept@anotherdb d
WHERE e.deptno = d.deptno
;
```

Формально связи с БД являются хранимыми объектами (создаются и удаляются командами CREATE и DROP, наблюдаемы в таблице USER_DB_LINKS), но по отношению к объектам прочих категорий обладают редким свойством: их имена могут быть изначально составными, наподобие PRODUCT.OFFICE.MAIN, ORCL.CLASS и так далее.

Как и большинство хранимых объектов, связи с БД принадлежат конкретным схемам, однако права на обращения через них другим пользователям не передаются. Этим связи отличаются от более привычных объектов хранения, таких как таблицы, представления и от некоторых других. Расширение области употребления достигается иначе, а именно указанием при создании связи слова PUBLIC. Связи, создаваемые с указанием PUBLIC, доступны всем пользователям вообще, так как определяются вне схем, на уровне базы данных:

```
CREATE PUBLIC DATABASE LINK db.class
  USING 'oraserv:1521/orcl.class'
;
```

(Создать общедоступную связь с БД можно только при наличии привилегии CREATE PUBLIC DATABASE LINK.)

Эта одновременно и пример недоопределенной связи, которая далее может быть уточнена в отдельных схемах при том, что в каждой схеме указание имени пользователя и пароля допускается произвольно:

```
CONNECT scott/tiger
CREATE DATABASE LINK db.class
  CONNECT TO scott
  IDENTIFIED BY tiger
;
CONNECT system/oracle
CREATE DATABASE LINK db.class
  CONNECT TO yard
  IDENTIFIED BY pass
;
```

Такой техникой доопределения общедоступной связи в собственных схемах пользуются, в частности, при организации репликации данных с целью усиления защиты доступа, и при импорте данных в свою БД из посторонней.

Oracle может неправильно обрабатывать общедоступные связи с совпадающей первой составляющей частью в имени, например А и А.В. То есть, прежде чем создавать связь А, проверьте, не существуют ли уже связи вида А.В, и наоборот.

Есть ограничение на число возможных открытых связей на другую БД в пределах сеанса. Оно задается статическим параметром СУБД OPEN_LINKS, его умолчательное значение равно 4, и переопределить его вправе администратор БД.

11.4. Подпрограммы

Хранимыми программными единицами в Oracle являются процедуры и функции (общее название — подпрограммы), триггерные процедуры, пакеты, типы данных (учитывая программную логику их методов). Это объекты хранения в БД типов PROCEDURE, FUNCTION, TRIGGER, PACKAGE BODY, TYPE BODY.

Для обращения к подпрограммам (самостоятельным или в составе пакета) средствами SQL с версии 9 Oracle используется специальный оператор CALL (заимствован из ANSI/ISO SQL):

```
SQL> SET SERVEROUTPUT ON
SQL> CALL DBMS_OUTPUT.PUT_LINE ( 'This is a procedure call' );
```

Пример обращения оператором CALL к функции:

```
SQL> VARIABLE s NUMBER
SQL> CALL sys.standard.sin ( 1 ) INTO :s;
```

Call completed.

```
SQL> PRINT s
```

```
_____ S
.841470985
```

В SQL*Plus первый пример даст тот же результат, что и:

```
SQL> EXECUTE DBMS_OUTPUT.PUT_LINE ( 'This is a procedure call' )
```

Это равносильно выдаче:

```
SQL> BEGIN DBMS_OUTPUT.PUT_LINE ( 'This is a procedure call' ); END;  
2 /
```

Последняя форма более универсальна, так как она употребима в программах на включающих языках. В то же время, если необходимо обратиться к процедуре, и не более того, то оператор CALL окажется несколько более производителен, нежели блок BEGIN ... END.

Второй пример, с обращением к SIN, в SQL*Plus можно переинициализировать так:

```
SQL> EXECUTE SELECT sin ( 1 ) INTO :s FROM dual
```

PL/SQL procedure successfully completed.

```
SQL> PRINT s
```

```
          S  
-----  
.841470985
```

или сразу (но уже не специфично для SQL*Plus):

```
SQL> SELECT sin ( 1 ) FROM dual;
```

```
      SIN(1)  
-----  
.841470985
```

Процедуры, в отличие от функций, не могут употребляться в составе выражений в операторах DML. Создание процедур, функций, пакетов, а также триггерных процедур и типов (в полном объеме) относится к теме программирования Oracle с помощью PL/SQL.

11.5. Индексы

11.5.1. Индексы в БД в Oracle

Индексы (от латинского «указатель», обращающий внимание на запрещенные католической церковью книги или их фрагменты) являются хранимыми вспомогательными объектами, используемыми при обработке запросов на SQL. В базах данных смысл индексов — воплотить функцию, позволяющую по значению (или набору значений) получить адреса строк таблицы с этими значениями. В промышленных БД всех типов подобные функции обеспечиваются с помощью индекса как специальной хранимой структуры, хотя умозрительно требования в такой структуре нет, а нужна лишь функциональность.

Наиболее употребимы в Oracle *B*-деревоподобные* индексы. Они могут создаваться:

- автоматически, СУБД — как средство проверки ограничений целостности «первичный ключ» и «уникальность» в таблицах,
- либо вручную, разработчиком — ради ускорения доступа к строкам таблицы.

Во втором случае («вручную») для создания индексов используется специальная команда SQL.

Примеры:

```
CREATE INDEX emp_idx ON emp ( ename );
```

```
CREATE INDEX emp_desc_idx ON emp ( ename DESC );
```

```
CREATE UNIQUE INDEX name_loc_idx ON dept_copy ( dname, loc );
```

На выбор столбцов для древовидного индекса есть ограничения.

- Разрешено создавать индекс не более, чем на 32 столбцов.
- Нельзя индексировать столбцы некоторых типов (например, семейства LOB или же LONG/LONG RAW).

Влияние индексов на эффективность работы с БД противоречиво. Индексы:

- способны сокращать время обращения к строке таблицы,
- способны увеличивать время обращения к строке таблицы (при «неудачном» распределении индексированных строк по блокам),
- требуют места в БД,
- способны замедлять обновления таблиц.

Ускорение обращения к строке таблицы определяется, вопреки широко распространенному мнению, не избирательностью запроса и не размером таблицы, а в конечном итоге количеством посещаемых блоков. Этот показатель не всегда легко контролируется.

Некоторые общие и простые соображения по поводу использования древовидных индексов:

- Индекс неэффективен при малом количестве различных индексированных значений (например пол: «М» и «Ж»), когда они представлены примерно равными количествами.
- При отсутствии значений (NULL) сразу во всех индексруемых столбцах (если индекс построен по нескольким столбцам) строка не индексируется. Поиск «по отсутствующим значениям» будет игнорировать индекс и выполняться полным просмотром таблицы.

Второй по важности тип индекса появился в версии Oracle 8.1 и существует для Enterprise Edition. Это *поразрядный* (bitmap) индекс. Он используется исключительно для ускорения доступа к данным таблицы и дает отдачу во вполне определенных обстоятельствах.

Доменный индекс (иначе — прикладной, предметный) программируется разработчиком приложения для конкретного типа объектов, однако несколько видов доменных индексов приходит в готовом виде с ПО Oracle, будучи уже запрограммированными разработчиками СУБД.

Для всех видов индексов допускаются частные случаи конфигурации.

11.5.2. Индексы для проверки объявленных ограничений целостности

Употребление индексов ради ускорения доступа составляет предмет оптимизации запросов и не является темой настоящего материала. Здесь приводятся некоторые сведения об индексах, используемых СУБД для технической поддержки ограничений целостности.

При обычном объявлении в таблице первичного ключа или свойства уникальности столбцов СУБД автоматически создаст служебный уникальный древовидный индекс. В случае многостолбцовой уникальности допускается задать несколько ограничений на одних и тех же столбцах, но обязательно перечисляемых в разном порядке. Для всех таких ограничений будет использоваться один и тот же индекс — соответствующий первому по порядку создания ограничению. В результате следующих действий два «разных» ограничения АВ и ВА будут внутренне проверяться одним и тем же индексом АВ:

```
CREATE TABLE t ( a NUMBER, b NUMBER, c NUMBER );
```

```
ALTER TABLE t ADD CONSTRAINT ab UNIQUE ( a, b );
```

```
ALTER TABLE t ADD CONSTRAINT ba UNIQUE ( b, a );
```

В автоматику создания служебного индекса можно вмешаться. Так, желаемые свойства автоматически создаваемому индексу можно сообщить, вложив в предложение CREATE TABLE или ALTER TABLE ... ADD *ограничение* (где формулируется ограничение целостности) конструкцию CREATE INDEX, например:

```
CREATE TABLE t (
  c NUMBER PRIMARY KEY USING INDEX ( CREATE INDEX pk_t ON t ( c ) )
, d VARCHAR2 ( 100 )
```

);

Таким образом мы получаем возможность самостоятельно не только назвать индекс первичного ключа, но и указать свойства организации и хранения индекса (выше этого не сделано, если не считать задания программистом индексу имени на свое усмотрение).

Если на необходимые столбцы индекс был заведен ранее, в качестве служебного можно взять уже имеющийся:

```
CREATE TABLE t ( c NUMBER );
```

```
CREATE INDEX pk_t ON t ( c );
```

```
ALTER TABLE t ADD PRIMARY KEY ( c ) USING INDEX pk_t;
```

Это позволяет заметно снизить затраты на создание указанных ограничений для больших таблиц.

В последнем предложении конструкцию USING INDEX можно было бы не употреблять. Однако же если бы индекса PK_T заранее не существовало, эту же конструкцию можно было использовать для заведения индекса с желаемыми характеристиками, применив следующую формулировку:

```
... USING INDEX [имя_индекса] [свойства_индекса] ...
```

или даже:

```
... USING INDEX ( CREATE INDEX имя_индекса [свойства_индекса] ) ...
```

Обратите внимание, что индекс в этом случае не обязан быть уникальным. (*Упражнение.* Проверить свойство уникальности у индекса PK_T). Более того, если ограничение создается как DEFERRABLE, индекс *обязан* быть *неуникальным*, и именно таковым он при том создается СУБД автоматически.

Если при заведении ограничения используется существующий индекс, явно создаваемый или же создаваемый ради ограничения с возможностью отложенной проверки (DEFERRABLE), то в отличие от автоматического он не будет удаляться вместе с удалением ограничения. (*Упражнение.* Проверить это.) В этом есть своя логика, но это же может приводить к недоразумениям, когда по удалению ограничения целостности в БД сохранится «остаточный» индекс, не всегда нужный по делу.

Индекс на первичный ключ и на уникальные столбцы существует всегда благодаря автоматизму своего создания. С другой стороны, индекс на внешний ключ сам не создается и при необходимости делать это нужно вручную:

```
CREATE TABLE tr ( d NUMBER REFERENCING t ( c ) );
```

```
CREATE INDEX fk_t ON tr ( d );
```

```
DROP INDEX fk_t;
```

Упражнение. Проверить, что создание индекса на внешний ключ не оказывает влияния на логику поведения последнего.

Решение о создании индекса на столбцы внешнего ключа принимается исходя из конкретных обстоятельств.

11.6. Таблицы с временным хранением строк

Отличаются от обычных таблиц БД тем, что время хранения *строк* в них ограничен концом либо транзакции, либо сеанса связи с СУБД — по выбору разработчика БД. *Описания* же таких таблиц (метаданные) хранятся в словаре-справочнике БД на общих основаниях с описаниями обычных таблиц, то есть вплоть до выдачи команды DROP TABLE. Эти свойства объясняют выбор фирмой Oracle названия: GLOBAL TEMPORARY в отличие от таблиц LOCAL TEMPORARY, имеющих со времен SQL-92 (но не в Oracle), полный жизненный цикл которых ограничен программным блоком.

Пример создания таблицы с временем хранения строк, ограниченным транзакцией:

```
CREATE GLOBAL TEMPORARY TABLE temp AS SELECT * FROM emp WHERE 1 = 2;  
-- строк нет
```

```

INSERT INTO temp SELECT * FROM emp;
-- строки появились

SELECT * FROM temp;
-- проверка

COMMIT;
-- строки пропали

SELECT * FROM temp;
-- проверка

```

Упражнение. Как изменится результат команды CREATE выше, если в формулировке SELECT опустить фразу WHERE ?

Примеры создания таблиц с явно указанным временем хранения строк — до конца текущего сеанса или же до конца текущей транзакции:

```

CREATE GLOBAL TEMPORARY TABLE tx ( c NUMBER ) ON COMMIT PRESERVE ROWS;

CREATE GLOBAL TEMPORARY TABLE ts ( c NUMBER ) ON COMMIT DELETE ROWS;

```

ON COMMIT DELETE ROWS не требует явного указания, так как подразумевается по умолчанию.

Если не считать «короткого» времени жизни строк, по своим потребительским свойствам таблицы с временным хранением строк почти не отличаются от обычных. Например для них можно строить индекс (напомним: ведь их описание хранится постоянно); с ними можно связывать триггерные процедуры или представления (view).

Таблицы обоих видов предоставляют каждому сеансу собственное множество строк, независимое от строк, заведенных в других сеансах. (Для таблиц, где время хранения строк ограничено сеансом, это неочевидно). Однако выполнение операций DDL с такими таблицами СУБД по понятным причинам увязывает с наличием в них строк (собственных) в других сеансах. Так, построить индекс (CREATE INDEX) удастся только, если в данный момент другой сеанс не завел в таблице собственные строки. Таким образом косвенная связь содержимого таких таблиц в разных сеансах все-таки имеется.

Подобные таблицы используются не для моделирования прикладных данных, а как технологическое средство построения приложения: например, для хранения промежуточных результатов вычислений.

11.7. Таблицы с внешним хранением данных

В версии 9 в Oracle введены таблицы «с внешним хранением данных», представляющие собой по сути табличную видимость в БД данных, на деле хранимых в файлах ОС. Дословное название объектов этой категории — «внешние таблицы», что не точно отражает существо дела. Описания таких таблиц хранятся в справочной части БД на общих основаниях и столь же постоянно, как и описания обычных таблиц, а вот местом расположения данных назначаются не сегменты в табличном пространстве БД, а внешние файлы. Связь между таблицей, как объекта в БД, и файлами осуществляется через «драйвер» одного из двух видов:

- ORACLE_LOADER: способен отображать текстовые и прочие данные из файла ОС в таблицу;
- ORACLE_DATAPUMP^[10-]: допускает выгрузку данных БД в двоичный файл ОС и последующее многократное прочитывание их в виде таблицы.

^[10-] Начиная с версии 10.

Оба драйвера основаны на разной программной логике, соответственно, загрузчика данных (SQL*Loader) и перекачке, или же «насоса» данных (Oracle Data Pump). Как следствие, они предполагают разные форматы содержимого файлов с данными и влекут различия в синтаксисе команды CREATE TABLE. Так, синтаксис одной из приводимых ниже команд SQL создания таблицы отчетливо перекликается с синтаксисом оформления программного файла для загрузчика Oracle.

11.7.1. Таблицы с драйвером ORACLE_LOADER загрузчика данных

Для таблиц, построенных с помощью драйвера ORACLE_LOADER, исходные данные берутся чаще всего из текстового файла. С версии 11.2, кроме того, в определении таких таблиц разработчику разрешено сослаться на «препроцессор», то есть на свою собственную программу преобразования, так что исходные данные уже не обязаны иметь текстовый вид.

Далее приводится пример одностороннего отображения «файл → таблица» с использованием драйвера загрузчика данных. Подразумевается, что действия выполняются на сервере (единственное место, где это используется — при правке содержимого файла с исходными данными командами HOST в SQL*Plus и *echo* командной оболочки ОС).

Предположим наличие каталога EXTFILES_DIR в БД, доступного для чтения (READ). Создадим файл *employee.txt* с исходными данными:

```
Bush,13/03/2001,5000
Powell,14/03/2001,,400.50
```

Обратите внимание, что во второй строке применено английское форматирование записи числа. Оно предполагает ту языковую привязку СУБД к местности, где это форматирование принято. Например, это соответствует паре язык/территория со значениями AMERICAN/AMERICA.

Заведем в схеме SCOTT таблицу с внешним хранением:

```
CREATE TABLE emp_load
( ename    VARCHAR2 ( 10 )
, hiredate DATE
, sal      NUMBER ( 7, 2 )
, comm     NUMBER ( 7, 2 )
)
ORGANIZATION EXTERNAL
( TYPE ORACLE_LOADER
  DEFAULT DIRECTORY extfiles_dir
  ACCESS PARAMETERS
  ( RECORDS DELIMITED BY NEWLINE
    NOBADFILE
    NOLOGFILE
    FIELDS TERMINATED BY ','
    MISSING FIELD VALUES ARE NULL
    ( ename
    , hiredate CHAR DATE_FORMAT DATE MASK "dd/mm/yyyy"
    , sal
    , comm
    )
  )
  LOCATION ( 'employee.txt' )
)
;
```

Проверка в SQL*Plus:

```
SQL> SELECT * FROM emp_load;
```

ENAME	HIREDATE	SAL	COMM
Bush	13-MAR-01	5000	
Powell	14-MAR-01		400.5

```
SQL> HOST echo Hussein,,1000.44,20000 >> employee.txt
```

```
SQL> /
```

ENAME	HIREDATE	SAL	COMM
Bush	13-MAR-01	5000	
Powell	14-MAR-01		400.5
Hussein		1000.44	20000

```
SQL> SELECT SUM ( sal ) FROM emp_load;
```



```

SUM(SAL)
-----
6000.44

```

Во втором запросе результат вычисляется по ходу прочтения файла *employee.txt*, то есть не требуется полной загрузки данных этого файла в БД для вычисления результата.

Конструкция LOCATION в определении таблицы допускает задание списка файлов или же переопределение файлов-источников, например:

```

SQL> HOST echo Laden,13/03/2005,10000 > centralasia.txt

SQL> ALTER TABLE emp_load LOCATION ( 'employee.txt', 'centralasia.txt' );

Table altered.

SQL> SELECT * FROM emp_load;

```

ENAME	HIREDATE	SAL	COMM
Bush	13-MAR-01	5000	
Powell	14-MAR-01		500.5
Hussein		1000.44	20000
Laden	13-MAR-05	10000	

Это позволяет переориентировать таблицу EMP_LOAD на другие файлы по мере подготовки их внешними программами приложения. Переопределять разрешено и DEFAULT DIRECTORY.

Еще пример свойств. Указания NOBADFILE и NOLOGFILE можно заменить на противоположные (и подразумеваемые молчаливо), в результате чего в каталоге начнут появляться протокольные файлы доступа к данным из БД. Это, однако, потребует дополнительно привилегии WRITE для SCOTT на использование каталога EXTFILES_DIR (для предыдущих действий хватало привилегии READ). Указание REJECT LIMIT сообщит предельное количество игнорируемых нарушений формата в записях из файлов-источников, обнаруживаемых в процессе выполнения SELECT:

```

CONNECT / AS SYSDBA
GRANT WRITE ON DIRECTORY extfiles_dir TO scott;

CONNECT scott/tiger

ALTER TABLE emp_load ACCESS PARAMETERS (
    RECORDS DELIMITED BY NEWLINE
    BADFILE
    LOGGING
);

ALTER TABLE emp_load REJECT LIMIT 20;

```

Здесь, пока нарушений формата количеством менее 21-го, СУБД не будет порождать ошибку доступа, а вместо этого информацию о нарушениях будет заносить в протокольный файл.

Таблицу с внешним хранением можно использовать для обновления данных наряду с обычными:

```

MERGE INTO bonus b USING emp_load e
ON ( b.ename = e.ename )
WHEN MATCHED THEN UPDATE SET sal = sal * 10
;

```

11.7.2. Таблицы с драйвером ORACLE_DATAPUMP перекачки данных

Таблицы с драйвером ORACLE_DATAPUMP заполняют файл данными автоматически в момент своего создания, и дальше разрешают выборку данных произвольное число раз. В БД будет храниться только описание таблицы, но не ее данные. Тем самым предоставляется возможность создавать в разные моменты времени снимки содержимого одной и той же таблицы, и обращаться к этим снимкам, не требуя для них места в БД и ресурсов буферизации блоков в СУБД. Если перенести файл с экспортированными данными на другую установку, их можно будет читать в табличной форме и в другой БД.

Тем самым может достигаться частичное решение задачи сохранения данных на логическом уровне и их переноса.

Примеры создания таблицы с внешним хранением данных с применением драйвера ORACLE_DATAPUMP:

```
CREATE TABLE ext_emp
ORGANIZATION EXTERNAL (
  TYPE ORACLE_DATAPUMP
  DEFAULT DIRECTORY extfiles_dir
  LOCATION ( 'emp_export.dmp' )
)
AS SELECT * FROM emp
;
```

В результате приведенной команды в каталоге ОС, на который ссылается каталог Oracle по имени EXTFILES_DIR, появится файл с данными. Формат файла закрытый и понятен только драйверу ORACLE_DATAPUMP, однако описательная часть его представляет собою текст в формате XML.

```
SQL> SELECT COUNT ( * ) FROM ext_emp;

COUNT(*)
-----
      14
```

Обращаться к таблице EXT_EMP командой SELECT можно без ограничений.

Пример, показывающий, как таблицы подобного рода можно использовать для переноса данных путем копирования файла экспорта (в нашем случае это *emp_export.dmp*):

```
CREATE TABLE ext_emp2 (
  empno    NUMBER ( 4 )
, ename    VARCHAR2 ( 10 )
, job      VARCHAR2 ( 9 )
, mgr      VARCHAR2 ( 10 )
, hiredate DATE
, sal      NUMBER ( 7, 2 )
, comm     NUMBER ( 7, 2 )
, deptno   NUMBER ( 2 )
)
ORGANIZATION EXTERNAL (
  TYPE ORACLE_DATAPUMP
  DEFAULT DIRECTORY extfiles_dir
  LOCATION ( 'emp_export.dmp' )
);
-- проверка:
SELECT COUNT ( * ) FROM ext_emp2;
```

Потребительскую ценность подобных таблиц увеличивает, сверх приведенных основных свойств, ряд дополнительных. Так, допускается ссылка на несколько файлов экспорта:

```
ALTER TABLE ext_emp2 LOCATION ( 'emp_export.dmp', 'emp_export.dmp' );
```

Проверка:

```
SQL> SELECT COUNT ( * ) FROM ext_emp2;

COUNT(*)
-----
      28
```

Файлы не обязаны находиться в общем каталоге, но ссылку на неумолчательный каталог следует обозначить явно, например:

```
ALTER TABLE ext_emp2
  LOCATION ( 'emp_export.dmp', my_work_dir:'emp_export.dmp' )
;
```

Создание и чтение данных можно распараллелить, что особенно выгодно при наличии нескольких файлов с данными:

```
ALTER TABLE ext_emp2 PARALLEL 2;
```

Примеры указания «предела терпимости» количества ошибок доступа к данным и отключения протоколирования ошибок доступа:

```
ALTER TABLE ext_emp2 REJECT LIMIT 10
;
ALTER TABLE ext_emp2 ACCESS PARAMETERS ( NOLOGGIN )
;
```

С версии 11.2 разрешено также при создании подобных таблиц задать уплотненное хранение данных в файлах и шифровать содержимое:

```
CREATE TABLE ext_emp3
ORGANIZATION EXTERNAL (
  TYPE ORACLE_DATAPUMP
  DEFAULT DIRECTORY extfiles_dir
  LOCATION ( 'emp_export3.dmp' )
  ACCESS PARAMETERS ( COMPRESSION ENABLED
                        ENCRYPTION ENABLED )
)
AS SELECT * FROM emp
;
```

Замечание. Шифрование в этом случае выполняется средствами TDE («прозрачного шифрования данных»), а оно требует наличия бумажника (Oracle wallet), находящегося в открытом состоянии во время выполнения действий по шифрованию и дешифрованию.

12. Некоторые общие свойства объектов хранения разных видов

Формально в Oracle имеется несколько десятков разных видов хранимых в БД объектов. Некоторое представление о многообразии дает запрос:

```
SELECT DISTINCT object_type FROM all_objects;
```

(Не все из них управляются командами SQL CREATE/ALTER/DROP, значительная часть — процедурно).

Некоторые группы видов объектов хранения объединены общими свойствами. Например, переименование объекта командой RENAME выполняется для таблиц, представлений данных, генераторов последовательности и для частных синонимов. О подобных общих свойствах говорится ниже.

12.1. Пространства имен для объектов в Oracle

Для именования объектов хранения Oracle разных видов используются различные пространства имен. Распределение по пространствам имен для наиболее популярных типов поясняется таблицей.

<i>Отдельное общее пространство имен</i>	<i>Отдельные собственные пространства имен</i>
Таблицы Представления данных Генераторы последовательностей из чисел Частные синонимы Хранимые процедуры Хранимые функции Пакеты Материализованные представления данных Собственные типы пользователей ...	Индексы Объявленные ограничения целостности Кластеры Триггерные процедуры Частные связи с иной БД Каталог в ОС Публичные синонимы Публичные связи с иной БД ...

Например, *разрешено* назвать в одной схеме *одним и тем же именем* таблицу, индекс, ограничение целостности и каталог. При работе со схемой SCOTT следующие команды не вызовут ошибок:

```
CREATE UNIQUE INDEX emp ON emp ( ename )  
;  
-- Ошибки нет.  
ALTER TABLE emp ADD CONSTRAINT emp UNIQUE ( ename ) USING INDEX emp  
;  
-- Ошибки нет.
```

Не разрешено назвать в одной схеме *одним и тем же именем* таблицу и представление данных, таблицу и функцию и так далее. При работе со схемой SCOTT следующие команды вернут в программу ошибку:

```
CREATE VIEW emp AS SELECT * FROM emp  
;  
-- Ошибка !.  
CREATE SEQUENCE emp  
;  
-- Ошибка !
```

12.2. Редакции объектов БД в Oracle

С версии 11.2 для некоторых видов хранимых объектов в Oracle можно заводить разные «редакции» (editions) и переключаться между ними в работе, моделируя одновременное наличие нескольких версий

прикладного программного обеспечения при его разработке или переделке. Речь не идет о «редакциях данных», и на таблицы такая техника не распространяется. Она применима к объектам следующих видов:

- VIEW
- SYNONYM
- PROCEDURE
- FUNCTION
- TRIGGER
- PACKAGE/PACKAGE BODY
- TYPE/TYPE BODY
- LIBRARY.

Основное применение техники редакций объектов можно видеть в области и поддержки и развития приложения. Она позволяет в некоторых случаях выполнять часть работ по внесению изменений в существующее прикладное ПО и в БД без останова использования уже существующей системы и отлаживать необходимые нововведения в параллель основной работе.

Создание редакций конкретных объектов сопряжено с определенными ограничениями. Скажем, нельзя создавать публичный синоним на редакцию объекта (к примеру, на редакцию какой-нибудь функции или какого-нибудь представления).

В версии Oracle 11.2 техника редакций объектов воплощена в своем начальном варианте, предполагающем дальнейшее развитие. В версии 12 отдельные объекты схемы, объявленной как редактируемой, можно заявлять нередатируемыми.

12.2.1. Создание редакций для объектов и управление ими

Управление редакциями регулируется привилегиями CREATE/ALTER/DROP ANY EDITION. Слово ANY в названиях напоминает о внесхемном характере редакций: они создаются на уровне БД вообще. Формально редакции приписаны пользователю SYS.

Если права создания редакций объектов и управления ими требуется выдать пользователю YARD, администратору БД следует выдать:

```
CONNECT / AS SYSDBA
GRANT CREATE ANY EDITION, DROP ANY EDITION TO yard;
```

Узнать действующую в данный момент редакцию можно из контекста сеанса USERENV (он встроен в СУБД и не требует для обращения к своим значениям никаких особых полномочий):

```
SQL> CONNECT yard/pass
Connected.
SQL> SELECT SYS_CONTEXT ( 'USERENV', 'CURRENT_EDITION_NAME' ) FROM dual;

SYS_CONTEXT('USERENV','CURRENT_EDITION_NAME')
-----
ORAS$BASE
```

ORAS\$BASE — это встроенная в БД и умолчательно действующая редакция, опираясь на которую администратор может создавать последовательность (а в будущих версиях Oracle, возможно, дерево) собственных редакций. Имя умолчательно установленной для БД редакции можно прояснить запросом:

```
SELECT property_value
FROM database_properties
WHERE property_name = 'DEFAULT_EDITION'
;
```

Примеры создания редакций:

```
CREATE EDITION app_release_1;
CREATE EDITION app_release_2 AS CHILD OF app_release_1;
-- и так далее
```

В первом случае редакция APP_RELEASE_1 была создана на основе исходной умолчательной редакции ORA\$BASE, во втором — как следует из текста команды.

Откомментировать редакцию в словаре-справочнике БД можно командой COMMENT:

```
COMMENT ON EDITION app_release_1
  IS 'The first release of application'
;
```

Снимается комментарий, как обычно, указанием пустой строки ". Просмотр комментариев выполняется через таблицу ALL_EDITION_COMMENTS.

Узнать существующие редакции в их взаимосвязи можно запросом к особой таблице:

```
SQL> SELECT * FROM all_editions;
```

EDITION_NAME	PARENT_EDITION_NAME	USA
ORA\$BASE		YES
APP_RELEASE_1	ORA\$BASE	YES
APP_RELEASE_2	APP_RELEASE_1	YES

Удалить из дерева (пока что — ветки) редакций можно только лист, свободный от подчиненных редакций:

```
DROP EDITION app_release_2;
```

Для того, чтобы пользователь Oracle мог не просто обращаться с редакциями объектов, но и формировать их, ему следует сообщить особое качество:

```
CONNECT / AS SYSDBA
ALTER USER yard ENABLE EDITIONS;
```

Качество ENABLE EDITIONS не изначальное, и неотъемлемое; буде оно раз выдано, *отменить его нельзя*. В результате все пользователи Oracle оказываются разделены на две категории: тех, кому разрешено формировать (строить) редакции, и тех, кому не разрешено. При том, возможен перевод пользователя из второй категории в первую, но никак не обратно. Удостовериться в наличии свойства ENABLE EDITIONS у пользователя можно по значению столбца EDITIONS_ENABLED (нового в версии 11.2) в таблице DBA_USERS (ее владелец SYS, и обычным пользователям она сама по себе не видна).

После выдачи последней команды каждый объект пользователя YARD, для которого разрешено создавать редакции, так или иначе будет привязан к какой-нибудь из нескольких редакций.

12.2.2. Настройка на работу с нужной редакцией

Чтобы пользователь Oracle имел право в конкретном сеансе работать с конкретной редакцией:

- (1) он должен иметь привилегию на работу с редакцией, выданную лично ему или, вместо этого, псевдопользователю PUBLIC (то есть всем вообще);
- (2) сеанс должен быть переключен на работу с этой редакцией.

Выдать пользователю личное общее разрешение на работу с объектами требуемой редакции можно примерно так:

```
GRANT USE ON EDITION app_release_1 TO scott;
```

USE — это привилегия на объекты вида EDITION, передаваемая кроме того через PUBLIC и через роли. Если редакцию, объявить в БД, как умолчательную, она автоматически полагается выданной для PUBLIC, то есть общедоступной, и не требует личных (или же ролевых) разрешений. По этой причине изначально частных разрешений на работу с ORA\$BASE не требуется — оно есть у всех. То же самое случится с редакцией APP_RELEASE_1, если в какой-то момент выдать:

```
ALTER DATABASE DEFAULT EDITION = app_release_1;
```

На последнюю команду способен обладатель привилегии ALTER DATABASE (а ею сейчас обладают SYS и SYSTEM, но пока не YARD). Как только такая команда будет выдана, выполнять GRANT USE, как выше, для придания нужных полномочий пользователю SCOTT, не потребуется. Выдачей подобной команды может венчаться процесс отладки новых редакций объектов («перевод приложения на новую редакцию»).

После получения разрешения (то есть привилегии) на работу с объектами конкретной редакции, пользователь Oracle приобретает право в рамках *отдельных* сеансов настраиваться на нее:

```
SQL> CONNECT scott/tiger
Connected.
SQL> SELECT SYS_CONTEXT ( 'USERENV', 'CURRENT_EDITION_NAME' ) FROM dual;

SYS_CONTEXT('USERENV','CURRENT_EDITION_NAME')
-----
ORA$BASE

SQL> ALTER SESSION SET EDITION = app_release_1;

Session altered.

SQL> SELECT SYS_CONTEXT ( 'USERENV', 'CURRENT_EDITION_NAME' ) FROM dual;

SYS_CONTEXT('USERENV','CURRENT_EDITION_NAME')
-----
APP_RELEASE_1
```

Код выше подтверждает то, что при открытии сеанса «сама собою» действует редакция, объявленная ранее умолчательной в БД.

12.2.3. Пример создания и использования разных редакций представления данных (view)

К настоящему моменту в БД образовались две редакции. Будем формировать их содержание редакциями объектов в схеме YARD. Создадим в последней две несложные редакции одного и того же представления данных — с выдачей сведений об отделе сотрудника, и без:

```
CONNECT yard/pass
ALTER SESSION SET EDITION = ora$base;
CREATE OR REPLACE EDITIONING VIEW empl
AS
  SELECT empno, ename, deptno FROM emp
;
ALTER SESSION SET EDITION = app_release_1;
CREATE OR REPLACE EDITIONING VIEW empl
AS
  SELECT empno, ename FROM emp
;
```

Настройку на редакцию ORA\$BASE можно было выше не выполнять, потому что эта редакция умолчательная (это проверялось ранее), и автоматически действует в начале каждого сеанса.

В результате появились две редакции представления данных EMPL:

```
SQL> SELECT view_name, edition_name FROM user_editioning_views_ae;
```

VIEW_NAME	EDITION_NAME
EMPL	ORA\$BASE
EMPL	APP_RELEASE_1

Редактируемые представления данных (editioning views) отличаются от обычных не только формальным словом EDITIONING при создании, но и некоторыми техническими свойствами. Они могут строиться на основе единственной таблицы, без фильтрации строк фразой WHERE и с отсутствием преобразований столбцов (в то же время, воспроизведение всех столбцов не обязательно). Есть и другие отличия, не востребованные настоящим примером.

Привилегии доступа к данным в разных редакциях выдаются по-отдельности:

```
ALTER SESSION SET EDITION = ora$base;
GRANT SELECT ON empl TO scott;
ALTER SESSION SET EDITION = app_release_1;
GRANT SELECT ON empl TO scott;
```

Вот как ниже SCOTT этими привилегиями пользуется:

```
SQL> CONNECT scott/tiger
Connected.
SQL> ALTER SESSION SET EDITION = ora$base;

Session altered.

SQL> SELECT * FROM yard.empl WHERE ROWNUM = 1;
```

EMPNO	ENAME	DEPTNO
7369	SMITH	20

```
SQL> ALTER SESSION SET EDITION = app_release_1;

Session altered.
```

```
SQL> SELECT * FROM yard.empl WHERE ROWNUM = 1;
```

EMPNO	ENAME
7369	SMITH

Теперь без отмены прежнего представления данных (и без прекращения работы с ним текущего приложения) открылась возможность независимо отлаживать приложение применительно к новому.

Упражнение. Отобрать у пользователя SCOTT привилегию на выборку данных из YARD.EMPL в редакции APP_RELEASE_1 и наблюдать результат попытки обращения.

12.2.4. Методология использования в связи с изменением структуры таблиц

Хотя техника редакций объектов хранения не распространяется на данные в таблицах БД, подготовить приложение к переходу на новые структуры имеющихся таблиц иногда помогают версии представлений. Фирма Oracle в своей документации приводит пример подобного употребления редакций. Идея этого примера излагается ниже.

Пусть требуется изменить структуру таблицы EMP, например, добавить новый столбец. Это может быть вызвано желанием заменить столбец ENAME на два столбца: отдельно для имени сотрудника и отдельно для фамилии. Загодя отладить имеющееся приложение для новой структуры можно следующим образом.

Во-первых создать на основе таблицы представление, воспроизводящее полностью данные таблицы. Командами RENAME подменить имя таблицы на искусственное, а старое EMP передать представлению. Приложение от такой подмены не пострадает.

Вообще, в основе такого подхода к перестройке схемы лежит временное переключение с основных таблиц на работу с виртуальными.

Далее, добавить в таблицу новый столбец таким образом, чтобы представление продолжало предъявлять старые данные.

Создать новую редакцию представления, учитывающую новый столбец в таблице.

С этого момента приложение может продолжать работать с данными через первую редакцию представления и одновременно отлаживаться применительно ко второй редакции. Когда решено, что приложение отлажено, командами RENAME возвращаем таблице прежнее имя EMP и отказываемся от технологических представлений.

13. Некоторые замечания по оптимизации выполнения предложений SQL

Оптимизация выполнения предложений на SQL не является темой настоящего материала. Тем не менее общая логическая схема обработки запросов SQL дает шанс в некоторых случаях подобрать такую формулировку запроса, которая способна позволить разработчику СУБД предложить более выгодный по сравнению с другими формулировками план обработки. Иногда такая выгодная формулировка может сопровождаться некоторым изменением смысла запроса. Программист обязан это понимать и быть уверенным в правомерности замены формулировки в конкретных обстоятельствах приложения.

Реляционная модель избыточна в том смысле, что допускает наличие *разных* выражений над отношениями, дающих *одинаковый* результат. Оптимизацию выполнения предложений SQL напрямую не затрагивает, полагая выбор той или иной формулировки запроса делом удобства программиста и передавая задачу построения оптимального плана обработки исключительно в компетенцию СУБД. Тем не менее в порядке помощи разработчикам в рамках реляционной теории были предложены способы ускорить вычисления ответов на запросы. Этому служит техника равносильных преобразований. Однако подобная техника не в полной мере применима к SQL ввиду отличий модели данных, подразумеваемой этим языком, от модели отношений (реляционной).

SQL предполагает возможное влияние формулировки запроса на эффективность вычислений. Кроме этого он явно вводит специальные структуры в БД для ускорения вычислений: это индексы и овеществленные представления данных (materialized views). Многие советы по формулировкам запросов в SQL ставят своей целью побудить СУБД использовать при доступе к данным индекс.

Oracle при формировании плана обработки поступившего запроса пытается переформулировать запрос в более выгодный для вычислений вид. При этом используются шаблоны равносильных преобразований, наличие вспомогательных структур БД и статистика объектов хранения. Сверх этого Oracle дает такие средства влияния на схему вычисления, как подсказки оптимизатору, особые параметры СУБД и особые конфигурации структур хранения объектов.

(Сказанное не относится к рекурсивным запросам, с CONNECT BY и с вынесенным подзапросом. Теоретически они инородны в Oracle SQL, техника равносильных преобразований к ним не применима, и СУБД иногда попросту не в силах предложить для них алгоритм обработки, отличный от механически простых вычислений. Примером может служить упоминавшийся ранее запрос порождения таблицы чисел с CONNECT BY LEVEL <= число.)

Оптимизатор вычисления запросов в Oracle не всегда терпит вмешательство со стороны программиста. Нередко он самостоятельно дает нетривиально разумные планы обработки. Однако в других случаях он порой в состоянии удивить незаmysловатостью своих решений и игнорированием очевидных с человеческой точки зрения наблюдений. Поэтому вопрос о целесообразности попыток воздействовать на работу оптимизатора в конкретных запросах пока еще не закрыт.

Приводимые ниже формулировки запросов в некоторых случаях следуют сразу нескольким рекомендациям. Рекомендации в целом носят качественный характер, в то время как количественная оценка выигрыша есть предмет отдельного и конкретного изучения.

13.1. Сокращение вычислений при локализации объектов доступа

Полное указание имен объектов в запросе пусть незначительно, но сокращает время разбора. Например, есть два запроса:

```
SELECT emp.ename FROM scott.emp;
```

```
SELECT ename FROM emp;
```

Второй имеет предпосылки обрабатываться дольше, так как при разборе запроса требует дополнительной работы по уточнению принадлежности таблицы схеме и столбца таблице. Заметьте к тому же, что, строго говоря, эти предложения не равносильны: при разборе второго может оказаться, что таблица EMP не принадлежит пользователю SCOTT (если запрос выдавался другим пользователем, имеющим одноименную таблицу, или если переменная сеанса CURRENT_SCHEMA имела значение, отличное от SCOTT).

13.2. Отказ от повторных вычислений выражений

В некоторых случаях формулировка запроса позволяет отказаться от повторного вычисления выражений. Примерами могут служить логические выражения BETWEEN и IN. Например, формулировка условного выражения:

```
( sal + comm ) BETWEEN 1000 AND 2000
```

способна обрабатываться эффективнее равносильной формулировки:

```
( sal + comm >= 1000 ) AND ( sal + comm <= 2000 )
```

Ответ на вопрос, будет ли первая формулировка *действительно* обрабатываться быстрее, зависит от сложности выражения. В данном случае подвыражение (SAL + COMM) достаточно просто, чтобы при построении плана, на этапе анализа формулировки, оптимизатор заметил, что во втором случае подвыражение повторяется, и фактически вычислял бы его однократно. Однако если бы вместо этого подвыражения стояла более сложная конструкция или если бы подвыражение содержало бы обращение к функции пользователя (все аспекты вычисления которой оптимизатору неизвестны), формулировку условного выражения через BETWEEN следовало бы признать более выгодной с вычислительной точки зрения. (Другие выгоды от BETWEEN — в надежности кода, из-за однократности *записи* выражения в запросе и в возможности задействовать в плане вычисления индекс, когда таковой имеется).

Аналогичные рассуждения обосновывают выгоду от использования для построения условных выражений операторов IN или = ANY перед употреблением цепочек сравнения, составленных с помощью OR, и выгоду от ссылки на имя столбца, данное во фразе SELECT, перед воспроизведением выражения в другой фразе. Так, возвращаясь к одному из примеров выше, предпочтение следует отдать формулировке:

```
SELECT job, AVG ( sal ) avgsal FROM emp
GROUP BY job
ORDER BY avgsal
;
```

перед формулировкой:

```
SELECT job, AVG ( sal ) FROM emp
GROUP BY job
ORDER BY AVG ( sal )
;
```

Опять-таки, что касается вычислений, то для этого простого случая обе формулировки скорее всего приведут к одному общему сценарию обработки (это не совсем просто проверить), но когда вместо AVG (SAL) будет стоять более сложное выражение или оно будет содержать обращение к функции пользователя, бесспорно предпочтительней окажется формулировка первого типа.

13.3. Оптимизация вычисления составного логического выражения

Логические выражения, составленные цепочками с помощью связок OR или AND, при отсутствии у Oracle информации о сложности вычисления подвыражений вычисляются во вполне определенном направлении и до выяснения окончательного результата (экономно). Это обстоятельство можно

использовать для размещения элементов цепочек, наиболее вероятно дающих TRUE или же FALSE в соответствующих концах цепочки, или же менее трудоемких в вычислениях (заметьте, что трудоемкость вычисления функции в выражении может быть известна программисту, но не СУБД).

При просмотре сотрудников выражение:

```
( sal > 1000 ) OR ( mgr IS NULL )
```

будет вычисляться скорее, чем:

```
( mgr IS NULL ) OR ( sal > 1000 )
```

Это объясняется тем, что цепочка, построенная с помощью связок OR, вычисляется слева направо, а сотрудников с зарплатой более 1000 — больше, чем «президентов». Мизерная в данном случае разница может оказаться заметной на вычислительно более сложных выражениях.

13.4. Переформулировка для сокращения объема обрабатываемых строк

Перестановка фильтров строк способна сократить объем обработки. Так, предложение SELECT с фразой HAVING иногда можно переформулировать в содержательно равносильное, но вычислительно более эффективное за счет перенесения отбора строк из фразы HAVING во фразу WHERE. Вот пример запроса на количество разных сотрудников по специальностям, помимо клерков:

```
SELECT    job, COUNT ( * )
FROM      emp
GROUP BY  job
HAVING    job <> 'CLERK'
;
```

Из сказанного ранее следует, что логическая последовательность действий по вычислению результата на этот запрос будет следующей:

- 1) отбор строк по условию WHERE;
- 2) разбиение строк на группы по условию GROUP BY;
- 3) вычисление сверток (агрегатов) по каждой группе;
- 4) отбор требуемых групп по условию HAVING.

Памятуя логический порядок вычислений, можно «помочь» оптимизатору запросов в СУБД сократить объем вычислений затратной группировки, переписав запрос в виде:

```
SELECT    job, COUNT ( * )
FROM      emp
WHERE     job <> 'CLERK'
GROUP BY  job
;
```

Некоторые специалисты вовсе дают рекомендацию избегать отсева групп фразой HAVING, ровно по этой причине.

Подобный перенос фильтра на более раннюю фазу обработки иногда возможен и в иных случаях, например, в операциях соединения. Сравните:

```
SELECT e.ename, d.dname
FROM   emp e
INNER JOIN dept d
USING  ( deptno )
WHERE  e.job <> 'SALESMAN'
;
```

```

SELECT e.ename, d.dname
FROM
  ( SELECT * FROM emp WHERE job <> 'SALESMAN' ) e
INNER JOIN
  dept d
USING
  ( deptno )
;

```

В то же время не следует недооценивать оптимизатор Oracle: для двух последних запросов (пусть и не самых сложных) он дает одинаковый план исполнения, так что выбор одного из них становится лишь делом предпочтения программиста. Просто в одних случаях оптимизатор может вести себя очень грамотно, разумнее программиста, но зато в других — с точки зрения программиста явно нерационально, и не замечать «очевидных» программисту вещей.

13.5. Возможность использовать индекс для доступа к строкам таблицы

Хотя доступ к строкам таблицы по индексу не гарантирует высокую скорость (а иногда приводит даже к замедлению), часто он оказывается оправдан. Но решению СУБД употребить существующий индекс может, помимо прочего, препятствовать формулировка условного выражения. Например, в случае указания поля, соответствующего индексированному обычным индексом (древовидным и без функционального преобразования ключа) столбцу, в качестве параметра для функции СУБД откажется от использования индекса:

```

SELECT ename FROM emp WHERE TRUNC ( empno ) = 7369;

```

СУБД может воспользоваться индексом (если сочтет целесообразным), только если индексированное поле присутствует в одной из частей сравнения без каких-либо преобразований:

```

SELECT ename FROM emp WHERE empno >= 7369;

```

но не в этом случае:

```

SELECT ename FROM emp WHERE empno + 0 >= 7369;

```

Oracle не будет также отказываться от использования индекса в сравнениях с помощью других операторов:

```

empno BETWEEN 7000 AND 8000
empno IN ( 7369, 7865, 8888 )
empno = ANY ( 7369, 7865, 8888 )
empno = ALL ( 7369, 7865, 8888 )

```

В сравнении оператором LIKE СУБД сможет привлекать индекс только, если в проверочной маске первый слева символ не является специальным:

```

job LIKE 'SAL%'

```

но не:

```

job LIKE '_SAL%'

```

14. Транзакции и блокировки

Транзакции и блокировки есть механизм регулирования доступа к БД из приложений.

Транзакция в SQL есть логическая последовательность операций DML по внесению изменений в БД, принимаемая или же отвергаемая СУБД в конечном итоге («по завершению транзакции») в целом. В английском языке слово transaction обозначает единицу общения двух агентов (в нашем случае программы и СУБД), завершающуюся оказанием взаимного воздействия друг на друга.

Понятие транзакции в реляционной теории отсутствует и составляет самостоятельный по отношению к ней предмет изучения. В случае баз данных транзакции позволяют:

- восстанавливать данные при аварийном прекращении сеанса связи программы с СУБД («единица восстановления»);
- гарантировать целостное представление данных программе при одновременной работе с данными нескольких программ;
- гарантировать целостное хранение данных в БД при одновременной работе с ними нескольких программ.

Широко известны общие требования к механизму транзакций («свойства ACID»):

- *атомарность*: последовательность изменений в БД, поступающая из программы во время транзакции принимается или отвергается по завершению транзакции целиком, как единое целое;
- *согласованность (состоятельность)*: каждая транзакция переводит БД в новое согласованное (состоятельное) состояние, при том что, пока она не завершена, такового может и не быть;
- *изолированность*: изменения в БД, совершаемые в рамках транзакции, станут видны другим транзакциям только после ее фиксации;
- *долговечность*: изменения в БД, совершенные в рамках транзакции, не могут пропасть из-за сбоя работы СУБД.

Некоторые эксперты полагают, что требование согласованности для транзакций в БД, то есть соблюдение ограничений целостности, должно обеспечиваться на уровне не транзакции (как того допускают и стандарт SQL, и Oracle), а отдельного оператора DML. Получается, что выполнение этого требования механизмом транзакций свидетельствует о недостаточности языка SQL для моделирования событий в предметной области. К сожалению это не единственная возможная претензия к языку.

Стандарт SQL предлагает определенный перечень средств для управления транзакциями. Oracle не поддерживает их в полном объеме и в полной мере, однако средства для управления транзакциями в Oracle обладают свойствами ACID и достаточны для нужд большинства приложений.

14.1. Транзакции в Oracle

Для работы с транзакциями Oracle поддерживает следующие операторы SQL:

```
COMMIT [ WORK ]
```

```
ROLLBACK [ WORK ] [ TO SAVEPOINT имя_точки_сохранения ]
```

```
SAVEPOINT имя_точки_сохранения
```

```
SET TRANSACTION тип_транзакции
```

Слово WORK в COMMIT и ROLLBACK носит косметический характер и употребляется по желанию.

14.1.1. Команды COMMIT и ROLLBACK

В Oracle отсутствует команда для создания новой транзакции (в отличие от стандарта SQL), но есть две команды завершения: фиксацией результатов выполнявшихся в последней транзакции команд DML (COMMIT) и отказом от них (ROLLBACK). Соединение с СУБД автоматически приводит к началу новой транзакции, и то же случается по завершению отработки любой команды COMMIT или ROLLBACK. Таким образом все операции с данными (таблиц, индексов, внутренних объектов LOB) волей-неволей всегда выполняются в Oracle в рамках какой-нибудь транзакции, а сеанс связи программы с СУБД выглядит последовательностью сменяющих друг друга транзакций. Команды завершения транзакции затрагивают только операции DML, но в некоторых случаях СУБД порождает такие команды самостоятельно. Так, всякая команда DDL завершается неявной (скрытой) выдачей COMMIT; аварийный разрыв сеанса или возникновение исключительной ситуации на уровне программы сопровождается неявной выдачей ROLLBACK.

Пример:

```
CONNECT scott/tiger
-- открыта новая транзакция
INSERT INTO emp ( empno, ename ) VALUES ( 1111, 'BUSH' );
UPDATE emp SET ename = 'LADEN' WHERE empno = 1111;
ROLLBACK;
-- старая транзакция завершена отменой UPDATE и INSERT; открыта новая транзакция
```

Иногда приводят два методических правила по употреблению COMMIT:

- а) выдавать COMMIT как только представится возможным, и
- б) не выдавать COMMIT раньше необходимого.

Операция COMMIT затратна для СУБД и при особо частой выдаче может заметно тормозить работу СУБД. В версии 10 введена возможность ускоренного выполнения COMMIT. Для этого в команде можно указать ключевые слова BATCH («групповая фиксация»: запись о выдаче COMMIT заносится в буфер журнала в СУБД, но не провоцирует перенос журнальных записей в файл) или NOWAIT (СУБД начинает обрабатывать следующую команду сеанса, не дожидаясь фактического завершения отработки COMMIT):

COMMIT WRITE [BATCH | IMMEDIATE] [WAIT | NOWAIT]

Пример такого поведения можно наблюдать, прогнав в SQL*Plus с помощью сценарного файла следующий текст:

```
CONNECT / AS SYSDBA
SELECT dname, ora_rowscn FROM scott.dscn;
UPDATE scott.dscn SET dname = dname WHERE ROWNUM <= 2;
COMMIT;
STARTUP FORCE
SELECT dname, ora_rowscn FROM scott.dscn;
UPDATE scott.dscn SET dname = dname WHERE ROWNUM <= 2;
COMMIT WRITE BATCH NOWAIT;
STARTUP FORCE
SELECT dname, ora_rowscn FROM scott.dscn;
```

Здесь разрыв транзакции достигается форсированной перезагрузкой СУБД: STARTUP FORCE.

Любое из указаний BATCH или NOWAIT в команде COMMIT WRITE отменяет гарантию со стороны СУБД попадания последних изменений в БД в случае сбоя, невзирая на выдачу программой пользователя COMMIT.

Умолчательный способ отработки COMMIT при наличии указанных вариантов можно установить для всей СУБД (ALTER SYSTEM ...) и для отдельных сеансов (ALTER SESSION ...) параметрами СУБД:

```
COMMIT_WRITE[10]
COMMIT_LOGGING[11-)
COMMIT_WAIT[11-)
```

^[10] В версии 10.

^{[11-)} С версии 11.

Примеры:

```
ALTER SESSION SET COMMIT_WRITE = 'batch, nowait';
```

С версии 11:

```
ALTER SESSION SET COMMIT_LOGGING = 'batch';  
ALTER SESSION SET COMMIT_WAIT = 'force_wait';
```

14.1.2. Команды ROLLBACK и ROLLBACK TO SAVEPOINT

Команда SAVEPOINT позволяет поставить в последовательности команд DML поименованную «точку сохранения», к которой можно вернуться в рамках текущей транзакции с тем, чтобы дать ей с этого места новое продолжение:

```
CONNECT scott/tiger  
-- новая транзакция ...  
INSERT INTO emp ( empno, ename ) VALUES ( 1111, 'BUSH' );  
UPDATE emp SET job = 'PRESIDENT' WHERE ename = 'BUSH';  
SAVEPOINT try_new_employee;  
-- поставили точку сохранения TRY_NEW_EMPLOYEE  
UPDATE emp SET ename = 'LADEN' WHERE ename = 'BUSH';  
DELETE FROM emp WHERE ename = 'LADEN';  
SELECT ename FROM emp WHERE empno = 1111;  
-- сотрудник 'LADEN'  
ROLLBACK TO SAVEPOINT try_new_employee;  
-- вернулись к точке сохранения, отказавшись от последних DELELE и UPDATE  
SELECT ename FROM emp WHERE empno = 1111;  
-- сотрудник 'BUSH'  
UPDATE emp SET job = 'CLERK' WHERE ename = 'BUSH';  
ALTER TABLE emp ADD UNIQUE ( job );  
-- хотя ошибка, но операции DDL, а потому неявная выдача COMMIT и новая транзакция ...  
ROLLBACK;  
-- новая транзакция, и далее без ошибок  
SELECT ename FROM emp WHERE empno = 1111;  
DELETE FROM emp WHERE ename = 'BUSH';  
COMMIT;
```

Точки сохранения могут иметься во множестве, и не возбраняется использовать одно и то же имя несколько раз. Однако если имена в пределах транзакции совпали, то вернуться можно будет только к той, что выдана последней. Это следует учитывать при выборе имени очередной точки сохранения.

В некоторых типах СУБД нет точек сохранения, зато есть более развитый аппарат вложенных транзакций.

14.1.3. Команда SET TRANSACTION

Команда SET TRANSACTION позволяет в начале транзакции (точнее, до выдачи первого изменяющего данные оператора DML) назначить тип транзакции.

Транзакция любого типа в Oracle не сможет увидеть изменения незавершенных других транзакций (нет так называемых «грязных» транзакций).

Тип транзакции READ WRITE умолчательный и не требует явного указания командой SET TRANSACTION. Транзакция этого типа позволяет программе выдавать команды DML изменения данных и наблюдать их результат, как будто бы он непосредственно совершается в БД.

Задание типа READ ONLY дает начало «читающей» транзакции, в течение которой программа изолируется от изменений в БД (выполняемых другими транзакциями) и видит состояние БД на момент выдачи SET TRANSACTION READ ONLY.

«Читающие» транзакции полезны при составлении отчетов, когда программе нужны согласованные данные из нескольких таблиц базы. Однако попытка выполнить в них INSERT, UPDATE или DELETE приведет к ошибке.

Тип транзакции ISOLATION LEVEL SERIALIZABLE аналогичен READ ONLY, но не запрещает выполнять собственные операции INSERT, UPDATE или DELETE. Последнее дает программисту свободу действия по сравнению с READ ONLY, однако чревато риском для транзакции оказаться заблокированной. Кроме того, СУБД Oracle не всегда способна обрабатывать транзакцию типа ISOLATION LEVEL SERIALIZABLE с той же производительностью, что и типа READ ONLY.

Пример последовательности выдачи команд в SQL*Plus:

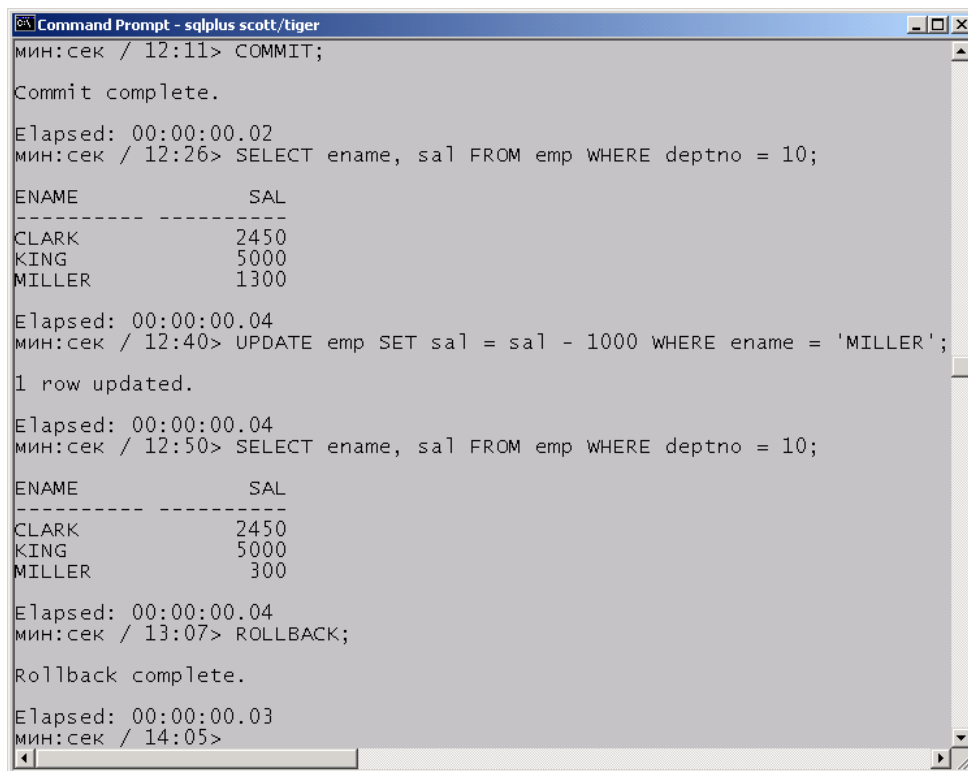
```
CONNECT scott/tiger
-- новая транзакция, по умолчанию — READ WRITE ...
INSERT INTO emp ( empno, ename ) VALUES ( 1111, 'BUSH' );
HOST sqlplus scott/tiger
    SELECT ename FROM emp WHERE empno = 1111;
    -- сотрудник не виден
    EXIT
COMMIT;
-- новая транзакция ...
SELECT ename FROM emp WHERE empno = 1111;
-- сотрудник 1111 находится в БД и виден
SET TRANSACTION READ ONLY;
-- установили тип READ ONLY ...
DELETE FROM emp WHERE empno = 1111;
-- ошибка: транзакция READ ONLY !
HOST sqlplus scott/tiger
    DELETE FROM emp WHERE empno = 1111;
    COMMIT;
    EXIT
SELECT ename FROM emp WHERE empno = 1111;
-- сотрудник все еще виден
ROLLBACK;
SELECT ename FROM emp WHERE empno = 1111;
-- ... а по завершению транзакции уже нет
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
-- установили тип ISOLATION LEVEL SERIALIZABLE ...
INSERT INTO emp ( empno, ename ) VALUES ( 1111, 'OBAMA' );
HOST sqlplus scott/tiger
    INSERT INTO emp ( empno, ename ) VALUES ( 2222, 'LADEN' );
    COMMIT;
    EXIT
SELECT ename FROM emp WHERE empno IN ( 1111, 2222 );
-- виден только 'OBAMA'
ROLLBACK;
SELECT ename FROM emp WHERE empno IN ( 1111, 2222 );
-- виден только 'LADEN'
DELETE FROM emp WHERE empno = 2222;
COMMIT;
-- «почистили» данные
```

Для удобства можно поменять умолчательный тип дальнейших транзакций в пределах текущего сеанса на желаемый, например:

```
ALTER SESSION SET ISOLATION LEVEL SERIALIZABLE;
```

14.2. Пример блокирования действий в транзакции

Oracle не запрещает разным транзакциям одновременно править разные строки одной и той же таблицы. Однако попытка транзакции изменить строку, уже изменяемую другой незавершенной транзакцией, приведет к блокировке претендующей транзакции. Ниже на двух экранах показана работа двух транзакций, сначала с разными строками EMP, а затем с общей строкой (текущее время отображается подсказкой для ввода строки в SQL*Plus):



```
Command Prompt - sqlplus scott/tiger
мин:сек / 12:11> COMMIT;

Commit complete.

Elapsed: 00:00:00.02
мин:сек / 12:26> SELECT ename, sal FROM emp WHERE deptno = 10;

ENAME          SAL
-----
CLARK           2450
KING            5000
MILLER          1300

Elapsed: 00:00:00.04
мин:сек / 12:40> UPDATE emp SET sal = sal - 1000 WHERE ename = 'MILLER';

1 row updated.

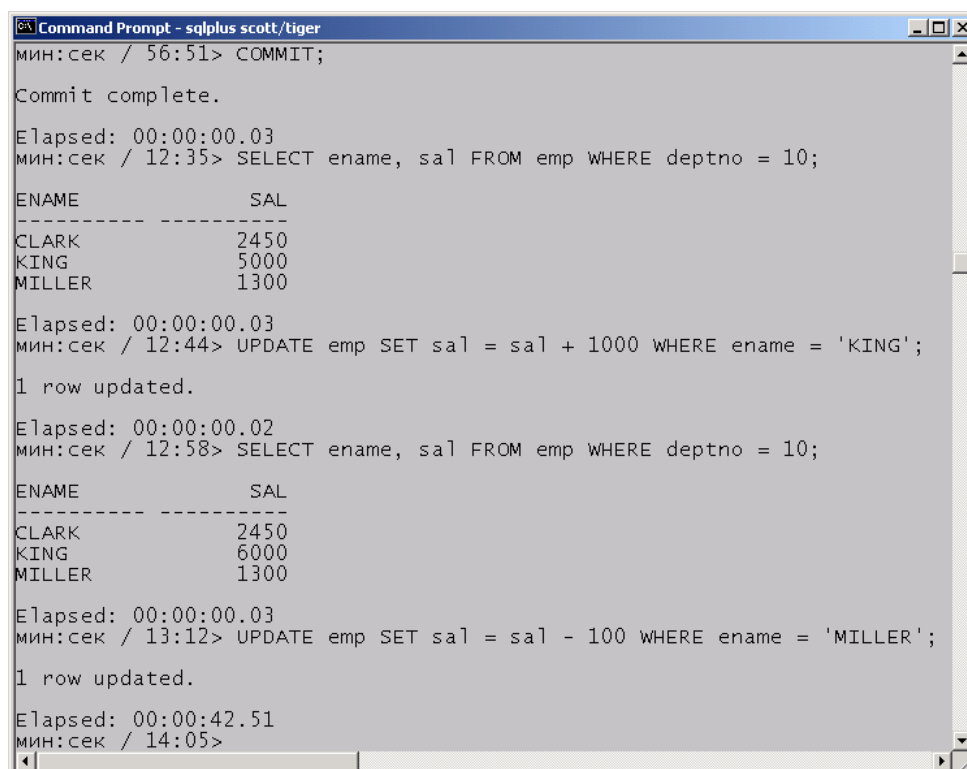
Elapsed: 00:00:00.04
мин:сек / 12:50> SELECT ename, sal FROM emp WHERE deptno = 10;

ENAME          SAL
-----
CLARK           2450
KING            5000
MILLER           300

Elapsed: 00:00:00.04
мин:сек / 13:07> ROLLBACK;

Rollback complete.

Elapsed: 00:00:00.03
мин:сек / 14:05>
```



```
Command Prompt - sqlplus scott/tiger
мин:сек / 56:51> COMMIT;

Commit complete.

Elapsed: 00:00:00.03
мин:сек / 12:35> SELECT ename, sal FROM emp WHERE deptno = 10;

ENAME          SAL
-----
CLARK           2450
KING            5000
MILLER          1300

Elapsed: 00:00:00.03
мин:сек / 12:44> UPDATE emp SET sal = sal + 1000 WHERE ename = 'KING';

1 row updated.

Elapsed: 00:00:00.02
мин:сек / 12:58> SELECT ename, sal FROM emp WHERE deptno = 10;

ENAME          SAL
-----
CLARK           2450
KING            6000
MILLER          1300

Elapsed: 00:00:00.03
мин:сек / 13:12> UPDATE emp SET sal = sal - 100 WHERE ename = 'MILLER';

1 row updated.

Elapsed: 00:00:42.51
мин:сек / 14:05>
```

Обратите внимание на «долгое» выполнение последнего оператора UPDATE на нижнем экране (42,51 секунды). В данном случае оно означает не то, что уменьшение зарплаты Миллеру на 100 единиц происходило так медленно, а то, что транзакция на нижнем экране после обращения к СУБД с заявкой на UPDATE была переведена в состояние ожидания (сеанс «завис»). Ее работу «заблокировала» транзакция на верхнем экране. Как только блокирующая транзакция завершилась, блокированная продолжила работу и на деле выполнила UPDATE. Об этой синхронизованности событий указывает (в подсказке SQL*Plus) окончательно одно и то же текущее время двух сеансов — для одной транзакции после ее последней команды ROLLBACK, а для другой после ее последней команды UPDATE.

Упражнение. Проверить возможность правки разными транзакциями (сеансами) одной строки таблицы, но разных полей.

Легко убедиться, что приведенная схема блокировок не препятствует появлению взаимных блокировок двух разных транзакций (deadlocks). Хорошая новость в том, что Oracle автоматически распознает появление взаимных блокировок и отменяет действие операции, виновной в этом. Одна из транзакций сможет продолжать работу и по своему окончанию освободит вторую.

Упражнение. Спровоцируйте взаимную блокировку двух транзакций и проверьте реакцию на это Oracle.

Показанное на примере блокирование действий транзакции, изменяющей данные, препятствует разрушению целостности данных в случае попыток одновременной правки со стороны разных программ. Технически этот механизм защиты данных строится на основе использования «замков». Строго говоря, Oracle не употребляет замки индивидуально для каждой строки, но для понимания логики происходящего можно пойти на такое допущение.

14.3. Замки, их типы, режимы наложения и правила совместимости

Замки (locks, иначе «блокировки») в Oracle являются средством предотвращения нежелательного одновременного доступа к данным и внутренним структурам СУБД путем либо выстраивания процессов СУБД в очередь, либо прерывания операции с возвращением в программу ошибку доступа.

Замки имеют *тип* (type) и *режим наложения* (mode).

Oracle использует несколько десятков различных типов замков, однако для регулирования изменений данных в таблицах первоочередную важность имеют замки всего двух типов, необходимость наличия которых на объекте доступа и обуславливает возможность выполнения изменяющей операции DML:

- TM: блокировка транзакцией целой таблицы («DML-блокировка»);
- TX: блокировка транзакцией отдельных строк таблицы («блокировка транзакции»).

Режимов наложения замка на объект шесть:

<i>Режим блокировки</i>	<i>Код — краткое название</i>	<i>(Иногда) иное название</i>	<i>Старое название</i>
NULL	1 — NL		
ROW SHARE	2 — RS	SUB SHARED (SS)	SHARE UPDATE
ROW EXCLUSIVE	3 — RX	SUB EXCLUSIVE (SX)	
SHARE	4 — S		
SHARE ROW EXCLUSIVE	5 — SRX	SHARED SUB EXCLUSIVE (SSX)	
EXCLUSIVE	6 — X		

Режимы с характеристикой EXCLUSIVE предполагают монопольное овладение объектом, а режимы с характеристикой SHARE — доленое, допускающее одновременное нахождение на объекте нескольких однотипных замков (для режима SRX первенствует поведение EXCLUSIVE). Эти две характеристики («виды блокировок») существуют в общем подходе к устройству транзакций. Слово ROW в названиях сообщает о действии замка на группу строк, а отсутствие этого слова — о действии на всю таблицу.

Замки типа TX могут налагаться СУБД в режимах RS и RX; типа TM — во всех.

Если транзакция пытается наложить на объект замок в режиме, несовместимом с режимом ранее наложенного на тот же объект замка, она либо (а) будет поставлена в очередь, либо (б) получит от СУБД сообщение об ошибке.
Правила совместимости режимов наложения замков:

Режим имеющейся блокировки	Режим претендующей блокировки					
	NL	RS	RX	S	SRX	X
NL	OK ^[1]	OK	OK	OK	OK	OK
RS	OK	(OK) ^[2]	(OK)	(OK)	(OK)	Несовм. ^[3]
RX	OK	(OK)	(OK)	Несовм.	Несовм.	Несовм.
S	OK	OK	Несовм.	OK	Несовм.	Несовм.
SRX	OK	OK	Несовм.	Несовм.	Несовм.	Несовм.
X	OK	Несовм.	Несовм.	Несовм.	Несовм.	Несовм.

^[1] режим нового замка совместим с режимом ранее наложенного, и замок будет применен

^[2] режим нового замка совместим с режимом ранее наложенного, если замок устанавливается командой LOCK TABLE и «условно совместим», если замок устанавливается командами UPDATE, DELETE и SELECT ... FOR UPDATE; в последнем случае транзакция встанет в очередь ожидания, если другие транзакции не заблокировали требуемые строки (TX).

^[3] режим нового замка несовместим с режимом ранее наложенного и попытка его наложить на объект приведет к ошибке.

Замки могут накладываться СУБД автоматически (неявно) и программой пользователя явочным порядком. Все замки, наложенные в течение транзакции как явно, так и неявно, автоматически снимаются по ее завершению. При возврате к точке сохранения (ROLLBACK TO SAVEPOINT ...) автоматически снимаются замки, наложенные в течение транзакции с момента выдачи команды создания точки сохранения.

14.4. Неявные блокировки при операциях DML

При поступлении из программы любой из команд INSERT, UPDATE или DELETE, СУБД сначала автоматически попытается наложить на объект определенные замки и только в случае успеха приступит к самому изменению данных. По меньшей мере, замков два:

- типа TM в режиме ROW EXCLUSIVE;
- типа TX в режиме EXCLUSIVE.

Наблюдать блокировки можно запросами SQL к таблицам словаря-справочника, но нагляднее их представляет Oracle Enterprise Manager (OEM — программа для администрирования Oracle). Возвращаясь к примеру выше, после первой выдачи UPDATE в рамках первой транзакции (она выполнялась сеансом 128) программа OEM показала следующие замки:

Username	Sessions Blocked	Session ID	Serial Number	Process ID	SQL ID	Lock Type	Mode Held	Mode Requested	Object Type	Object Owner	Object Name
User Locks											
SCOTT	0	128	3671	18533		TX	EXCLUSIVE	NONE	TABLE	SCOTT	EMP
SCOTT	0	128	3671	18533		TM	ROW EXCLUSIVE	NONE	TABLE	SCOTT	EMP

После попытки выполнить UPDATE для той же строки другой транзакцией (сеанс 139) эта транзакция «подвисла», а программа OEM показала следующие замки:

Username	Sessions Blocked	Session ID	Serial Number	Process ID	SQL ID	Lock Type	Mode Held	Mode Requested	Object Type	Object Owner	Object Name
User Locks											
SCOTT	1	128	3671	18533		TX	EXCLUSIVE	NONE	TABLE	SCOTT	EMP
SCOTT	0	139	1136	18531	dfu03ysq371mv	TX	NONE	EXCLUSIVE	TABLE	SCOTT	EMP
SCOTT	0	128	3671	18533		TM	ROW EXCLUSIVE	NONE	TABLE	SCOTT	EMP
SCOTT	0	139	1136	18531	dfu03ysq371mv	TM	ROW EXCLUSIVE	NONE	TABLE	SCOTT	EMP

Заметьте, что замки типа TM в режиме наложения ROW EXCLUSIVE не мешали друг другу, а попытка второй транзакции наложить замок типа TX в режиме EXCLUSIVE привела к постановке этой транзакции в очередь ожидания. Мешающий этому действию замок будет снят только по концу первой транзакции.

14.4.1. Влияние внешних ключей

Наличие внешних ключей в схеме обычно приводит к дополнительным замкам на объектах БД и к дополнительным шансам блокирования работы транзакций.

Если таблицы связаны внешним ключом, выполнение INSERT, UPDATE внешнего ключа одной таблицы или ключа (первичного или уникального) другой и DELETE для подчиненной таблицы автоматически добавит третий и, возможно, четвертый замок:

- типа TM в режиме ROW SHARE на партнерскую таблицу;
- типа TX в режиме EXCLUSIVE на партнерскую таблицу.

В некоторых версиях Oracle в подобном поведении возможны непринципиальные варианты.

При выполнении команды SELECT ... FOR UPDATE применительно к родительской таблице СУБД автоматически добавит второй замок:

- типа TM в режиме ROW SHARE на эту таблицу

Вот как OEM показывает замки, возникшие после выдачи UPDATE на изменение значения DEPTNO сотрудника из таблицы EMP:

Username	Sessions Blocked	Session ID	Serial Number	Process ID	SQL ID	Lock Type	Mode Held	Mode Requested	Object Type	Object Owner	Object Name
User Locks											
SCOTT	0	128	3671	18533		TX	EXCLUSIVE	NONE	TABLE	SCOTT	EMP
SCOTT	0	128	3671	18533		TX	EXCLUSIVE	NONE	TABLE	SCOTT	DEPT
SCOTT	0	128	3671	18533		TM	ROW EXCLUSIVE	NONE	TABLE	SCOTT	DEPT
SCOTT	0	128	3671	18533		TM	ROW EXCLUSIVE	NONE	TABLE	SCOTT	EMP

Обратите внимание на наложение двух «лишних» замков, уже на таблицу DEPT. Пример показывает, что полезные с точки зрения моделирования предметной области внешние ключи приводят к увеличению количества замков на объектах, а значит — повышают вероятность блокирования одними транзакциями других.

14.5. Явное наложение замка типа TM на таблицу командой LOCK TABLE

Используемая в Oracle техника замков позволяет предотвратить искажение данных изменяющими их транзакциями. Однако с точки зрения конкретной программы она может приводить к неожиданным «подвисаниям», смысл которых не всегда удастся осознать конечному пользователю (для него программа просто «перестает работать»). Чтобы не терять контроль над работой программы, разработчик может

побеспокоиться заранее и до действий по изменению строк явочным порядком наложить на объект требуемый замок. После этого транзакция сможет свободно изменять необходимые данные вплоть до своего завершения, не опасаясь быть заблокированной.

Предложение LOCK TABLE в Oracle позволяет наложить замок в требуемом режиме на таблицу:

```
LOCK TABLE имя_таблицы  
IN режим_блокировки MODE  
[NOWAIT | WAIT [ n ]]
```

Само по себе это предложение не полностью решает задачу сохранения контроля над работой программы, так как сама команда LOCK TABLE может столкнуться с несовместимым замком, повешенным ранее другой транзакцией. Для окончательного закрытия проблемы применяется указание NOWAIT/WAIT.

Указание NOWAIT заменит в случае несовместимости замков ожидание на немедленный возврат в программу сообщения об ошибке. Теперь уже дело программиста обработать такую ошибку (исключительную ситуацию, exception) и довести ее в понятном виде до конечного пользователя.

Указание WAIT соответствует умолчательному поведению, когда при занятости хотя бы одной строки таблицы другими транзакциями, выдающая LOCK TABLE будет ждать их завершения. Указание же WAIT *n* тоже приведет к ожиданию, но не более *n* секунд. Если за это время мешающий замок будет с таблицы снят, претендующая транзакция повесит на нее свой и продолжит работу. Если же по истечении *n* секунд таблица не освободится, программа получит сообщение об ошибке. Указание WAIT 0 равносильно NOWAIT.

Вариант WAIT *n* разрешен с версии 11.

Примеры:

```
LOCK TABLE dept IN SHARE MODE;  
  
LOCK TABLE emp, dept IN EXCLUSIVE MODE NOWAIT;
```

Блокировку таблицы в режимах SHARE и EXCLUSIVE можно специально запретить или же наоборот, разрешить (но не снять!). Пример:

```
ALTER TABLE emp DISABLE TABLE LOCK;
```

14.6. Явная блокировка групп строк в таблицах

Недостатком блокирования данных таблицы командой LOCK TABLE может оказаться чересчур широкий охват строк, способный порождать по существу ненужные ожидания среди прочих транзакций. Зарезервировать для собственных нужд группы строк, а не все строки таблицы целиком, можно оператором SELECT, завершив его специальной фразой FOR UPDATE.

Предложение:

```
SELECT ...  
FOR UPDATE [список_имен_столбцов]  
[[NOWAIT | WAIT [ n ]]
```

не только вернет в программу запрашиваемые данные, но и автоматически наложит два замка, связывающие с действиями в этой транзакции группы строк, участвующих в формировании результата запроса:

- типа TM в режиме ROW SHARE на эту таблицу, и
- типа TX в режиме EXCLUSIVE.

Примеры:

```
SELECT * FROM emp WHERE deptno = 10 FOR UPDATE;
```

Здесь программа получает сведения о сотрудниках 10-го отдела и тут же резервирует за собой право свободно их изменять до конца транзакции

Примечательно, что ограничений на однотабличность предложения SELECT не накладывается. Это позволяет одним предложением зарезервировать для последующих изменений группы строк сразу из нескольких таблиц, что важно, когда данные разных таблиц содержательно связаны:

```
SELECT empno, sal, comm
FROM   emp INNER JOIN dept
      USING ( deptno )
WHERE  job = 'CLERK'
      AND loc = 'NEW YORK'
FOR UPDATE
;
```

Здесь программа получает сведения о сотрудниках-клерках из Нью-Йорка и тут же блокирует строки об этих сотрудниках в таблице EMP и строке об отделе этих сотрудников из таблицы DEPT. В подобных случаях снова может возникать проблема чересчур широкого блокирования. Сузить охват блокирования искусственно позволяет уточнение OF ...:

```
SELECT empno, sal, comm
FROM   emp INNER JOIN dept
      USING ( deptno )
WHERE  job = 'CLERK'
      AND loc = 'NEW YORK'
FOR UPDATE OF emp.sal
;
```

Здесь программа получает те же сведения, что и в предыдущем запросе, но резервироваться до конца транзакции будут только строки из таблицы EMP с сотрудниками-клерками из Нью-Йорка. Строки из таблицы DEPT не блокируются.

Казалось бы, в уточнении OF ... достаточно сослаться на имена *таблиц*, строки которых мы хотим зарезервировать, в то время как Oracle требует указывать здесь список *столбцов* таблиц. Логика такого синтаксического оформления следующая: сообщить после слова OF перечень тех полей строк используемых в запросе таблиц, которые мы намерены править до конца транзакции. Таким образом фраза FOR UPDATE OF приобретает документирующий характер, способствующий лучшему восприятию текста программистом.

Указания NOWAIT и WAIT [n] даются и действуют аналогично таким же в команде LOCK TABLE, однако вариант WAIT n стал доступен много раньше, с версии 8.

14.6.1. Групповая блокировка исключительно свободных строк

С версии 8.1 существует разновидность групповой блокировки с обходом уже кем-то заблокированных в данный момент строк. Запрос ниже выдаст в программу и одновременно пометит для текущей транзакции замками только те строки о сотрудниках отдела 10, которые свободны в настоящее время от несовместимых замков, принадлежащих другим транзакциям:

```
SELECT * FROM emp WHERE deptno = 10 FOR UPDATE SKIP LOCKED;
```

Строки с существующими несовместимыми замками «наша» транзакция попросту проигнорирует, обойдет.

Такая возможность позволяет сократить и упростить программный код. Долгое время она была недокументирована, но с версии 11 стала официальной.

14.7. Другие замки, способные блокировать доступ к данным

14.7.1. Замки доступа, используемые предложениями DDL

На время выполнения предложений DDL ALTER TABLE, DROP TABLE и LOCK TABLE ... IN EXCLUSIVE MODE СУБД пытается «блокировать» таблицу замком типа TM в режиме EXCLUSIVE. Если там уже имеется замок от предшествовавшей операции DML, попытка выполнить операцию DDL может завершиться неудачей.

Упражнение. Проверить работу блокировок при выполнении операций DDL. Выполнить в одном сеансе (т. е. в одной транзакции):

```
UPDATE emp SET sal = sal WHERE ename = 'SCOTT';
```

Выполнить в другом сеансе (т. е. в другой транзакции):

```
DROP TABLE emp;
```

В тех случаях, когда приложения часто вносят несложные правки в таблицу, выдача команды DDL может чисто технологически превратиться в проблему. С версии 11 упростить дело позволяет параметр, задающий предельное время выжидания доступа к таблице для выполнения команды DDL. Параметр устанавливается администратором для СУБД, или же программистом для своего сеанса, например:

```
ALTER SESSION SET DDL_LOCK_TIMEOUT = 30;
```

```
DROP TABLE emp;
```

/ теперь если таблица EMP занята операцией DML, команда DROP TABLE будет ожидать ее освобождения в течение 30 секунд, и только если после этого таблица не успеет освободиться, вернет ошибку «ресурс занят» */*

В некоторых случаях блокирование данных таблиц можно приуменьшить, или даже его избежать. Например, при включении ранее отключенных ограничений PRIMARY KEY или UNIQUE, при наличии индекса, отказ от проверки на соответствия ограничению имеющихся данных с помощью слова NOVALIDATE выльется дополнительно в отсутствие блокирования данных таблицы:

```
ALTER TABLE emp MODIFY CONSTRAINT pk_emp ENABLE NOVALIDATE;
```

При включении ограничения с проверкой данных (VALIDATE, что по умолчанию) на время работы команды ALTER TABLE данные будут заблокированы для внесения изменений.

14.7.2. Замки доступа к версиям редакций объектов

Введенный в версии 11.2 механизм редакций объектов БД повлек за собой появление замков типа AX, накладываемых сеансом связи прикладной программы с СУБД. В этой версии они накладываются автоматически в режиме SHARE на редакцию ORA\$BASE. Обычно это не препятствует доступу, так как замки в режиме SHARE совместимы, однако в других случаях могут случиться блокировки.

15. Таблицы словаря-справочника

В основе «статических таблиц» словаря-справочника Oracle лежит относительно небольшое число *исходных* (основных) таблиц, таких, как OBJ\$, TAB\$, COM\$, IND\$, AUD\$, PROPS\$ и им подобных. О наличии этих таблиц в схеме SYS документация по Oracle не сообщает. Но на их основе построено большое число *выводимых* (виртуальных) таблиц, «представлений» данных, доставляющих справочную информацию об объектах в более удобном виде, собственно и предназначенных разработчиками Oracle для обычных потребителей (технически при помощи вдобавок одноименных таблицам публичных синонимов), например, таблицы:

```
USER (ALL, DBA) _TABLES
USER (ALL, DBA) _TAB_COLUMNS
USER (ALL, DBA) _INDEXES
USER (ALL, DBA) _CONSTRAINTS
USER (ALL, DBA) _CONS_COLUMNS
USER (ALL, DBA) _IND_COLUMNS
USER (ALL, DBA) _SEQUENCES
USER (ALL, DBA) _TRIGGERS
USER (ALL, DBA) _TRIGGER_COLS
...
```

Префикс USER обозначает перечисление объектов пользователя, префикс ALL — объектов, доступных для работы из схемы, префикс DBA — всех объектов БД. Таблицы с префиксом DBA обычному пользователю, как правило, не видны. В то же время некоторое количество таблиц-представлений словаря-справочника не имеют префиксов USER, ALL или DBA и именованы по-своему.

Так, таблица DICTIONARY является «справочной по справочнику»; она содержит список названий практически всех (виртуальных, view) таблиц словаря-справочника (за редкими исключениями) с краткими пояснениями — в том числе название самой себя. Она предназначена для поиска нужной справочной таблицы путем составления запроса на SQL, например:

```
SELECT table_name FROM dictionary WHERE table_name LIKE 'USER%DICT%';
```

Для простоты употребления заведено несколько дополнительных публичных (PUBLIC) синонимов:

- TABS (синоним для USER_TABLES) — список всех таблиц схемы пользователя;
- COLS (синоним для USER_TAB_COLUMNS) — список всех столбцов всех таблиц схемы пользователя;
- SEQ (синоним для USER_SEQUENCES) — список всех генераторов последовательностей чисел в схеме пользователя;
- SYN (синоним для USER_SYNONYMS) — список всех синонимов схемы пользователя;
- CAT (синоним для USER_CATALOG) — список основных таблиц и представлений данных, синонимов и генераторов последовательностей чисел в схеме пользователя;
- OBJ (синоним для USER_OBJECTS) — список всех объектов схемы пользователя;
- IND (синоним для USER_INDEXES) — список всех индексов схемы пользователя;
- DICT (синоним для DICTIONARY).

Введя в словарь-справочник представления данных, разработчики Oracle достигли сразу две цели:

- дали программистам БД более удобный вид на данные словаря-справочника, чем это обеспечивают исходные таблицы (а при проектировании последних приоритет отдавался критерию скорости доступа);
- уменьшили риски случайной порчи справочной части БД из-за некорректных изменений операторами DML напрямую справочных таблиц.

Когда пользователь работает в графической среде разработки типа SQL Developer, он редко испытывает нужду в прямом обращении к таблицам словаря-справочника: за него это скрытно делает программа. Однако даже в этом случае запросы к справочным таблицам на SQL способны иногда дать ответ, не доступный вовсе или получаемый крайне неудобно с помощью графической среды; например: «в каких таблицах имеется столбец с таким-то именем?».

16. Встроенный SQL

Техника встроенного SQL позволяет вставлять тексты запросов SQL в обычный язык программирования, называемый в этом случае включающим. Примеры включающих языков для Oracle: C/C++, Ada, COBOL, PL/1, FORTRAN, Java (SQLJ), SQL*Plus.

Пример оформления запроса SQL для включающего языка C/C++ (переменные привязки name, salary, d_no):

```
exec sql      select ename, sal
                into :name, :salary
                from emp
                where deptno = :d_no;
```

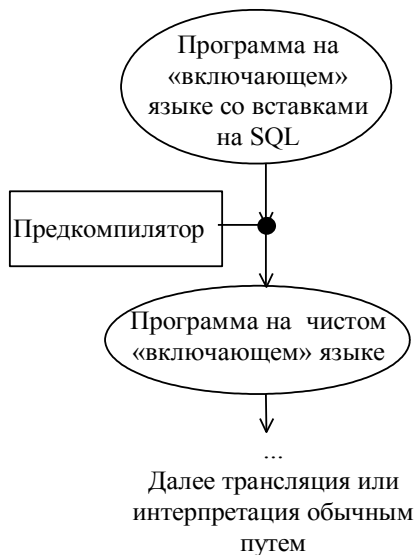
Пример для включающего языка Java (переменная привязки cnt):

```
#sql {
    SELECT COUNT ( * )
    INTO :cnt
    FROM emp
}
```

Пример из PL/SQL в окружении SQL*Plus (name, salary, d_no — переменные в SQL*Plus):

```
BEGIN
    SELECT ename, sal INTO :name, :salary
    FROM   emp
    WHERE  deptno = :d_no
    ;
END;
```

Порядок обработки исходного текста программы в общем случае:



Примеры предкомпиляторов для Oracle в таких случаях: Pro*C, Pro*Ada, Pro*PL/1, Pro*FORTRAN, SQLJ и др.

Некоторые примеры составления запросов

*Известное известно лишь немногим, а успех имеет одинаковый у
всех*

Аристотель. Поэтика

17. Запрос первых N записей

Запросы о «первых N записях» известны еще по реляционной теории БД, где они именовались «запросами с квотой», то есть, в переводе с латинского, с указанием ограничения объема возвращаемого результата. Такие запросы достаточно распространены в приложениях. Заметьте однако, что их смысл отличается от запросов с неполной выборкой строк из таблиц-источников, оформляемых в SQL с помощью конструкции SAMPLE после имени таблицы. Примеры запросов с пробной выборкой приводились в обсуждении предложения SELECT ранее.

17.1. Вопрос к БД

Примеры постановки вопроса. Имеющиеся данные:

ENAME	SAL
SMITH	800
ALLEN	1600
WARD	1250
JONES	2975
MARTIN	1250
BLAKE	2850
CLARK	2450
SCOTT	3000
KING	5000
TURNER	1500
ADAMS	1100
JAMES	950
FORD	3000
MILLER	1300

(1) Выдать пять сотрудников с наибольшими окладами. Должно быть:

ENAME	SAL
KING	5000
SCOTT	3000
FORD	3000
JONES	2975
BLAKE	2850

(2) Выдать пять сотрудников с наименьшими окладами. Должно быть:

ENAME	SAL
SMITH	800
JAMES	950
ADAMS	1100
MARTIN	1250
WARD	1250

(3) Выдать сотрудников с пятью наибольшими окладами. Должно быть:

ENAME	SAL
KING	5000
SCOTT	3000
FORD	3000
JONES	2975
BLAKE	2850
CLARK	2450

17.2. «Очевидное», но неправильное решение

```
SELECT  ename, sal
FROM    emp
WHERE    ROWNUM <= 5
ORDER BY sal DESC
;
```

Возможный ответ:

ENAME	SAL
KING	5000
SCOTT	3000
CLARK	2450
MILLER	1300
JAMES	950

Фраза WHERE, порождающая значения для ROWNUM, обрабатывается до ORDER BY. Поэтому приведенный выше запрос на деле узнает множество *произвольных* пяти сотрудников и предъявляет их в упорядоченном виде.

17.3. Правильные решения

Приводимое ниже решение годится для всех версий Oracle. SQL дает более одного способа строить такие запросы, но они могут иметь разные планы выполнения.

Приводимое решение рекомендуется для подобных случаев фирмой Oracle. Именно при таком составлении запрос специальным образом оптимизируется и *не* приводит к поочередно полному упорядочению всех строк таблицы EMP, а затем отбора первой пятерки, но вместо этого выльется лишь в *простой просмотр* этой таблицы, сопровождаемый незначительными накладными расходами:

```
SELECT *
FROM ( SELECT ename, sal FROM emp ORDER BY sal DESC )
WHERE ROWNUM <= 5
;
```

Запрос, оформленный по такому шаблону, дает единственный пример оправданности упорядочения строк в подзапросе — во всех остальных случаях упорядочение в подзапросе бессмысленно. Неочевидно, что эта оправданность — лишь логическая, а на деле Oracle будет делать так, как только что было сказано. Подобная формулировка дает очередное красноречивое доказательство отличия SQL от естественного языка с одной стороны, а с другой — несовпадения того, что записано, с тем, что выполняется в действительности.

Когда такие запросы устроены более сложно, как это бывает в жизни, их удобно оформлять с вынесением подзапроса во фразу WITH, на манер следующего:

```

WITH elist AS
    ( SELECT ename, sal FROM emp ORDER BY sal DESC )
SELECT *
FROM   elist
WHERE  ROWNUM <= 5
;

```

Другой схожий пример оформления текста запроса встречался ранее.

Когда в таких запросах желательно иметь в ответе не только строки, но и их расположение в N-ке, во фразу SELECT основного запроса следует добавить выдачу ROWNUM. Легко заметить, что в в нашем примере SCOTT и FORD получают при этом разные номера, хотя критерий их отбора, величина зарплаты, у них одинаков. Если в квотированном запросе требуется не пронумеровать строки, а выдать ранг (от немецкого Rang — класс, чин, разряд), потребуется воспользоваться аналитическими функциями RANK или DENSE_RANK, например:

```

WITH elist AS
    ( SELECT ename, sal, RANK ( ) OVER ( ORDER BY sal DESC ) FROM emp )
SELECT *
FROM   elist
WHERE  ROWNUM <= 5
;

```

Хотя здесь имеется обращение к аналитической функции, Oracle, обнаружив, что в окончательный ответ поступит ограниченное множество строк (WHERE ROWNUM <= 5), так же не будет тратить время на общее ранжирование, а будет вместо этого просматривать таблицу, выполняя по ходу дела минимум вычислений над небольшим массивом в оперативной памяти для очередной строки.

Наконец, версия 12 дала формулировку, напоминающую имевшиеся до этого у других производителей:

```

SELECT  ename, sal
FROM    emp
ORDER BY sal DESC
FETCH FIRST 5 ROWS ONLY
;

```

Внутри СУБД она обрабатывается техникой аналитической функции ROW_NUMBER, однако тема производительности здесь не рассматривается.

Упражнение. Построить запрос на выдачу сотрудников с пятой по счету сверху зарплатой.

18. Декартово произведение

Декартово произведение — одна из отправных операций над отношениями в реляционной модели. В SQL возникает или при отсутствии фразы для отбора WHERE вообще или же при отсутствии во фразе WHERE сравнения полей строк таблиц друг с другом. Примеры:

```
SELECT ename, empno, dname
FROM   emp, dept
;
```

```
SELECT ename, empno, dname
FROM   emp, dept
WHERE  dept = 10
;
```

Чаще всего (но не всегда) не имеет прикладного смысла и по этой причине является ошибкой приложения. Сложность в том, что СУБД не считает декартово произведение ошибкой и никогда программисту о его прикладной ошибке не сообщит. (Определенная подвижка произошла в Oracle версии 10, где администратор БД получил возможность анализировать группы запросов в том числе на предмет наличия декартовых произведений.)

Отличительная особенность декартова произведения — вероятный большой объем результата. Однако он может скрадываться дополнительной обработкой. Сравните, например, выдачу названий отделов и числа работающих сотрудников (при том, что один из запросов намеренно построен бессодержательно):

```
SELECT  dname, COUNT ( empno )
FROM    emp, dept
WHERE   dept.deptno = emp.deptno ( + )
GROUP BY dname
;
```

```
SELECT  dname, COUNT ( empno )
FROM    emp, dept
GROUP BY dname
;
```

Улавливать другие признаки, характерные для декартова произведения, как, например, затраченное время процессора, можно, но сложнее, чем объем строк.

Примечательно, что при систематическом использовании программистом синтаксиса SQL-92 для записи соединения (с версии 9 Oracle) появление подобных просчетов невозможно, так как этот синтаксис *требует* указания сравнения значений разных столбцов друг с другом (если конечно это не NATURAL INNER JOIN). На следующий запрос Oracle ответит сообщением о синтаксической ошибке:

```
SELECT dname, COUNT ( empno )
FROM   emp RIGHT JOIN dept
GROUP BY dname
;
```

Для случаев (— редких), когда декартово произведение все-таки нужно, предлагается специальная операция CROSS JOIN. Следующий запрос выполнится без ошибки:

```
SELECT dname, COUNT ( empno )
FROM   emp CROSS JOIN dept
GROUP BY dname
;
```

Последний запрос приведен чисто формально. Его прикладной смысл остается загадкой. В учебниках операцию декартового соединения иногда подкрепляют примером выдачи данных о рассылке писем ото всех всем.

19. Ловушка условия с отрицанием NOT

Понимание NOT в SQL не всегда интуитивно с точки зрения обыденной логики. Далее приводится пример расхожей ошибки неопытного программиста.

«Выдать отделы, где есть клерки»:

```
SELECT DISTINCT
  deptno
FROM
  emp
WHERE
  job = 'CLERK'
;
```

«Выдать отделы, где нет клерков» (*неправильно*):

```
SELECT DISTINCT
  deptno
FROM
  emp
WHERE
  NOT job = 'CLERK'
;
```

«Выдать отделы, где нет клерков» (*почти правильно*):

```
SELECT DISTINCT
  deptno
FROM
  emp
WHERE
  deptno NOT IN ( SELECT deptno FROM emp WHERE job = 'CLERK' )
;
```

«Выдать отделы, где нет клерков» (*совсем правильно*):

```
SELECT
  deptno
FROM
  dept
WHERE
  deptno NOT IN ( SELECT deptno FROM emp WHERE job = 'CLERK' )
;
```

Обратите внимание, что от простого обращения запроса запись на SQL существенно преобразилась и даже поменялась таблица перебора с EMP на DEPT !

Упражнение. Предложите другие корректные формулировки обратного запроса, например, с использованием MINUS. Предложите другую формулировку прямого запроса, более близкую по духу обратному.

20. Ловушка в NOT IN (S)

Непродуманное использование связки NOT IN (S) в SQL может приводить к противоречию с интуитивной логикой. Пусть, например, нужно выдать всех сотрудников, имеющих подчиненных. Естественно спросить:

```
SELECT
    ename
FROM
    emp
WHERE
    empno IN ( SELECT mgr FROM emp )
;
```

Пусть теперь нужно выдать сотрудников, не имеющих подчиненных. Если спросить:

```
SELECT
    ename
FROM
    emp
WHERE
    empno NOT IN ( SELECT mgr FROM emp )
;
```

то в ответе получим пустое множество. Чтобы понять, в чем дело, достаточно обратить внимание на отсутствие в столбце MGR значения (всегда ровно одного, если только в базе корректно отслеживается древовидная взаимосвязь сотрудников !) и вспомнить способ обработки конструкции NOT IN (S), приводившийся ранее. Для правильного результата строку с отсутствием значения в поле MGR следует отфильтровать:

```
SELECT
    ename
FROM
    emp
WHERE
    empno NOT IN ( SELECT mgr FROM emp WHERE mgr IS NOT NULL )
;
```

Чтобы не впасть в произвольную ошибку, рекомендуется избегать конструкции NOT IN (S). В нашем случае более четкой и эффективной в исполнении могла бы оказаться другая запись:

```
SELECT
    managers.ename
FROM
    emp managers
    LEFT OUTER JOIN
    emp employees
ON ( managers.empno = employees.mgr )
WHERE
    employees.mgr IS NULL
;
```

Содержательно она полностью равносильна предыдущей, но нечувствительна к возможным отсутствиям значений (т. е. не требует переформулировки в зависимости от того, есть или нет пропуски значений в столбце).

Обратите внимание, что по тому же типу можно переписать и прямой запрос:

```
SELECT DISTINCT
    managers.ename
FROM
```

```
emp managers
LEFT OUTER JOIN
emp employees
ON ( managers.empno = employees.mgr )
WHERE
    employees.mgr IS NOT NULL
;
```

Оба запроса в этой формулировке различаются (за вычетом необходимости в прямом запросе указать DISTINCT) лишь частицей NOT (что роднит их с соответствующими формулировками на естественном языке), но любопытно, что прямой запрос содержит ее, а обратный — нет !

Упражнение. Предложите еще одну корректную формулировку для прямого и обратного запросов, с использованием EXISTS.

21. Типичная ошибка в составлении полуоткрытых соединений

Полуоткрытые соединения в Oracle допускают две специальные формулировки: собственной, принятой фирмой, и основанной на обозначении '(+)', а также взятой из стандарта SQL:1999, основанной на обозначениях LEFT/RIGHT [OUTER] JOIN ... ON/USING. Собственная формулировка Oracle короче стандартной, и в некоторых (редких) случаях не заменима на стандартную, но также и чревата ошибками в использовании. Типичная проблема, связанная с неправильным употреблением '(+)' в цепочке полуоткрытых соединениях, рассматривается ниже.

Незавершенная цепочка полуоткрытых соединений может возникать при формировании последовательности соединений из трех и более таблиц. Схема возникновения ошибки:

<pre>SELECT ... FROM t1, t2, t3, t4 WHERE t1.c1 = t2.c1 (+) AND t2.c2 = t3.c2 (+) AND t3.c3 = t4.c3</pre>	<p><i>Должно быть</i></p> <p>→</p>	<pre>SELECT ... FROM t1, t2, t3, t4 WHERE t1.c1 = t2.c1 (+) AND t2.c2 = t3.c2 (+) AND t3.c3 = t4.c3 (+) или SELECT ... FROM t1 LEFT OUTER JOIN t2 USING (c1) t2 LEFT OUTER JOIN t3 USING (c2) t3 LEFT OUTER JOIN t4 USING (c3) или SELECT ... FROM t1 LEFT OUTER JOIN t2 ON t1.c1 = t2.c1 t2 LEFT OUTER JOIN t3 ON t2.c2 = t3.c2 t3 LEFT OUTER JOIN t4 ON t3.c3 = t4.c3</pre>
--	------------------------------------	--

Пример.

```
DROP TABLE j PURGE;
DROP TABLE e PURGE;
DROP TABLE d PURGE;
CREATE TABLE j
  ( job VARCHAR2 ( 9 ), maxsal NUMBER ( 7, 2 ) )
;
CREATE TABLE e
  ( ename VARCHAR2 ( 10 ), job VARCHAR2 ( 9 ), deptno NUMBER ( 2 ) )
;
CREATE TABLE d
  ( dname VARCHAR2 ( 14 ), deptno NUMBER ( 2 ) )
;

INSERT INTO j VALUES ( 'CLERK', 20000 );
INSERT INTO j VALUES ( 'SALESMAN', NULL );

INSERT INTO e VALUES ( 'SMITH', 'CLERK', 20 );
INSERT INTO e VALUES ( 'SCOTT', 'SALESMAN', 30 );
INSERT INTO e VALUES ( 'ALLEN', 'MANAGER', 20 );

INSERT INTO d VALUES ( 'SALES', 20 );
INSERT INTO d VALUES ( 'OPERATIONS', 30 );
INSERT INTO d VALUES ( 'ACCOUNTING', 40 );
```

Проверка:

```
SELECT d.dname, e.ename, j.maxsal
```

```

FROM    d, e, j
WHERE   d.deptno = e.deptno ( + )
AND     e.job    = j.job
;

```

Ответ:

DNAME	ENAME	MAXSAL
SALES	SMITH	20000
OPERATIONS	SCOTT	

```

SELECT d.dname, e.ename, j.maxsal
FROM    d, e, j
WHERE   d.deptno = e.deptno ( + )
AND     e.job    = j.job      ( + )
;

```

Ответ:

DNAME	ENAME	MAXSAL
SALES	SMITH	20000
OPERATIONS	SCOTT	
ACCOUNTING		
SALES	ALLEN	

Такой же ответ дает запрос, составленный в соответствии со стандартом (но уже без ошибочных вариантов):

```

SELECT d.dname, e.ename, j.maxsal
FROM
    d LEFT JOIN e USING ( deptno )
    LEFT JOIN j USING ( job )
;

```

Аналогично формулировка с ON вместо USING.

