Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Факультет информационных технологий и управления

Кафедра систем управления

М. А. Крупская

ОСНОВЫ ПРОГРАММИРОВАНИЯ С ИСПОЛЬЗОВАНИЕМ *SQL* И *PL/SQL*

Рекомендовано Министерством образования Республики Беларусь в качестве учебного пособия для студентов учреждений высшего образования по специальности «Системы управления информацией»

Минск БГУИР 2025

УДК 004.65(075.8)
ББК 32.972.134я73
К84

Р е ц е н з е н т ы:

кафедра технологий программирования
Белорусского государственного университета
(протокол № 14 от 21.03.2024);

и. о. заведующего кафедрой информационных систем и технологий
Белорусского государственного технологического университета,
старший преподаватель Е. А. Блинова

Крупская, М. А.

К84 Основы программирования с использованием *SQL* и *PL/SQL* : учеб. пособие / М. А. Крупская. – Минск : БГУИР, 2025. – 300 с. : ил.

ISBN 978-985-543-818-3.

Приведен теоретический материал, необходимый для получения базовых знаний и навыков работы с СУБД *Oracle* любому заинтересованному читателю. Основные синтаксические конструкции поясняются достаточным количеством практических примеров применения инструкций *SQL* и *PL/SQL* на моделях учебных баз данных, которые необходимо выполнять в ходе изучения курса. Каждый раздел заканчивается заданиями для самостоятельного выполнения или контрольными вопросами для проверки полученных знаний.

УДК 004.65(075.8)
ББК 32.972.134я73

ISBN 978-985-543-818-3

© Крупская М. А., 2025
© УО «Белорусский государственный университет информатики и радиоэлектроники», 2025

СОДЕРЖАНИЕ

Введение 6
1. Общие сведения и структура языка *SQL* 7
1.1. История развития СУБД *Oracle* 7
1.2. Основные продукты компании и редакции СУБД *Oracle* 11
1.3. Общие сведения о *SQL* 17
1.4. Основные правила написания *SQL*-операторов 20
Контрольные вопросы и задания к разделу 1 23
2. Операторы языка определения данных 24
2.1. Общие сведения об операторах *DDL* 24
2.2. Стандартные типы данных *SQL* и типы данных *Oracle* 26
2.3. Создание таблиц (инструкция CREATE TABLE) 31
2.4. Удаление таблицы (инструкции DROP TABLE и PURGE) 39
2.5. Изменение определения таблицы (инструкция ALTER TABLE) 40
2.6. Условия целостности данных 43
2.7. Создание представлений 50
2.8. Псевдонимы таблиц (инструкции CREATE/DROP SYNONYM) 56
2.9. Индексы (инструкции CREATE/DROP INDEX) 57
2.10. Последовательности (инструкции CREATE/DROP SEQUENCE) 59
Контрольные вопросы и задания к разделу 2 63
3. Операторы языка манипулирования данными 65
3.1. Общие сведения об инструкциях *DML* 65
3.2. Инструкция SELECT (общий синтаксис) 66
3.3. Предложения SELECT и FROM 69
3.4. Сортировка и ограничение строк в результате запроса 72
3.5. Отбор строк (предложение WHERE) 75
3.6. Многотабличные запросы 78
3.7. Стандартный синтаксис внутренних и внешних соединений 80
3.8. Предложения GROUP BY и HAVING 84
3.9. Статистические функции 87
3.10. Подчиненные запросы (подзапросы) 89
3.11. Добавление данных (инструкция INSERT INTO) 95
3.12. Удаление данных (инструкция DELETE) 99
3.13. Обновление данных (инструкция UPDATE) 101
Контрольные вопросы и задания к разделу 3 102
4. Транзакции 104
4.1. Понятие транзакции 104
4.2. Журнал транзакций и проблемы параллельной обработки 107
4.3. Блокировки транзакций 114
4.4. Понятие транзакции и виды блокировок в *Oracle* 119
4.5. Концепция многоверсионной модели согласованности по чтению 120
4.6. Управление транзакциями в *Oracle* 124
Контрольные вопросы и задания к разделу 4 126

5. Способы обеспечения безопасности данных 127
5.1. Принципы защиты данных в стандарте *SQL* 127
5.2. Разрешение и запрещение ролей в стандарте *SQL* 130
5.3. Аутентификация в *Oracle* 132
5.4. Профили в *Oracle* 136
5.5. Привилегии в *Oracle* 138
5.6. Роли в *Oracle* 147
5.7. Аудит и другие способы обеспечения безопасности в *Oracle* 150
Контрольные вопросы и задания к разделу 5 153
6. Общие сведения и структура языка *PL/SQL* 155
6.1. Преимущества языка 155
6.2. Лексические единицы 156
6.3. Структура блока 159
6.4. Раздел заголовка 160
6.5. Секция объявлений 165
Контрольные вопросы и задания к разделу 6 168
7. Типы данных *PL/SQL* 169
7.1. Скалярные типы 169
7.2. Составные типы 173
7.3. Работа с коллекциями 177
7.4. Ссылочные типы и типы больших объектов 183
7.5. Преобразование типов данных 185
7.6. Понятие курсоров. Явные курсоры 187
7.7. Неявные курсоры 191
7.8. Курсорные переменные и подзапросы 193
Контрольные вопросы и задания к разделу 7 197
8. Функциональные возможности программ 198
8.1. Условные управляющие операторы 198
8.2. Команды управления последовательностью выполнения 201
8.3. Команды цикла 203
8.4. Директивы компилятора 207
8.5. Автономные транзакции 208
8.6. Секция обработки исключений 209
Контрольные вопросы и задания к разделу 8 213
9. Программные единицы 214
9.1. Общие ключевые слова процедур и функций *Oracle* 214
9.2. Создание и выполнение процедур 215
9.3. Создание и вызов функций 217
9.4. Дополнительные возможности процедур и функций 220
9.5. Пакеты 224
9.6. Триггеры 227
Контрольные вопросы и задания к разделу 9 237
10. Стандартные пакеты 239
10.1. Пакет DBMS_SCHEDULER 239

10.2. Динамический *SQL*. Понятие привязки 243
10.3. Пакет DBMS_SQL 245
10.4. Обработка операторов DML в DBMS_SQL 249
10.5. Обработка запросов на извлечение информации в DBMS_SQL 251
10.6. Встроенный динамический *SQL* 255
10.7. Пакет DBMS_OUTPUT 258
10.8. Пакет UTL_FILE 262
10.9. Пакет DBMS_PIPE 269
Контрольные вопросы и задания к разделу 10 279
11. Хранилища данных 280
11.1. Концепция систем складирования данных 280
11.2. Архитектура хранилищ данных 284
11.3. Характеристика решений ведущих производителей 288
11.4. Типовые программно-аппаратные решения проектирования хранилищ данных 290
Контрольные вопросы и задания к разделу 11 297
Список использованных источников 299

ВВЕДЕНИЕ

В современном мире информация давно стала одним из ключевых ресурсов для любой организации или промышленного предприятия. Умение эффективно управлять информацией, обрабатывать и анализировать данные, принимать на их основе обоснованные решения – все это важные и необходимые навыки для любого специалиста.

Структура любой информационной системы может быть представлена совокупностью обеспечивающих подсистем, среди которых обычно выделяют информационное, техническое, математическое, программное, организационное и правовое обеспечение.

В целом информационное обеспечение – это совокупность информационных ресурсов и услуг, предоставляемых для решения управленческих и научно-технических задач в соответствии с этапами их выполнения. В автоматизированных системах информационное обеспечение представляет собой совокупность единой системы классификации и кодирования информации, унифицированных систем документации, схем информационных потоков, циркулирующих в организации, а также методологий построения баз данных (БД).

Пособие состоит из 11 разделов. Первые пять раскрывают основные возможности использования инструкций *SQL* как в их стандартной интерпретации, так и применительно к системе управления базами данных (СУБД) *Oracle*. Разделы с шестого по десятый – это погружение в возможности работы с процедурным языком СУБД *Oracle – PL/SQL*. Заключительный раздел посвящен хранилищам данных (ХД) как следующему этапу развития информационного обеспечения в современном мире и позволяет получить общее представление об информационных технологиях и принципах управления складированием данных.

Учебное пособие предназначено в первую очередь для тех студентов и молодых специалистов, кто хочет не просто понять принципы работы информационных систем управления, использующих для хранения и обработки информационного обеспечения реляционные и объектно-реляционные базы данных, а именно научиться на практике применять языки *SQL* и *PL/SQL* для решения конкретных задач, улучшить свои навыки алгоритмизации и программирования в области проектирования и реализации информационных систем, использующих реляционные БД на верхнем уровне систем управления. Представленный материал пособия предполагает, что у заинтересованных в его изучении лиц есть базовые знания и интересы в области общей теории проектирования реляционных и объектно-реляционных моделей данных, а также опыт разработки программ с использованием современных информационных технологий.

Необходимо отметить, что СУБД *Oracle* востребована в Беларуси и во всем мире из-за своих уникальных возможностей и функций, раскрыть которые помогает в том числе и это пособие, собравшее и адаптировавшее под конкретные задачи теоретические материалы из различных источников.

1. ОБЩИЕ СВЕДЕНИЯ И СТРУКТУРА ЯЗЫКА *SQL*

1.1. История развития СУБД *Oracle*

Корпорация *Oracle* (*Oracle Corporation*) – крупнейший в мире производитель программного обеспечения для организаций и поставщик серверного оборудования.

Корпорация *Oracle* была основана в 1977 году программистами компании *Ampex* Ларри Эллисоном (идейный вдохновитель и глава компании до 2014 года), Бобом Майнером и Эдом Оутсом. Основатели вложили в дело около 2000 долларов США и разработали СУБД для реляционных моделей БД, использовав язык *C* и *SQL*-интерфейс. Сначала компания носила название *Software Development Laboratories*, а в 1978 году была переименована в *Relational Software, Inc.*

В 1979 году выпущена первая официальная версия СУБД *Oracle* 2, работающая на мини-компьютере *PDP*-11 фирмы *Digital Equipment Corporation*, под управлением операционной системы (ОС) *RSX*-11 – она стала первой коммерческой системой управления реляционными БД на основе языка запросов *SQL*. Новую версию основатели назвали второй по маркетинговым соображениям – это название должно было сообщить заказчикам, что продукт *Oracle* проверен временем и сильно обогнал разработки компании *IBM*. СУБД *Oracle* получила широкое признание, так как могла обрабатывать огромные массивы данных даже на простых машинах.

В 1980 году выпущена версия *Oracle* 2 для ОС *VAX/VMS*.

В 1982 году выпущена версия *Oracle* 3, которая стала первой СУБД, поддерживающей обработку транзакций и способной выполняться на разных платформах, в том числе на мэйнфреймах и мини-ЭВМ. Даже *IBM* закупила для своих мэйнфреймов *S*/370 программу *Oracle*.

В 1983 году *Relational Software, Inc*. переименована в *Oracle Corporation*.

В 1984 году выпущена и перенесена на несколько различных платформ версия *Oracle* 4, поддерживающая как ОС *VAX*, так и *IBM VM*, и предоставляющая возможность стабильного пользовательского чтения данных. В это время СУБД *Oracle* появилась в России, во Всесоюзном научно-исследовательском институте системных исследований (ВНИИСИ) и до 1990-х годов продукт *Oracle* поставлялся в СССР нелегально.

В 1986 году выпущена версия *Oracle* 5, поддерживающая возможности создания приложений в архитектуре «клиент – сервер» (используя сетевой интерфейс *SQL*Net*), распределенную обработку данных, кластеры *VAX*. Эта версия стала одной из первых программ *MS DOS*, перешагнувших через 640 килобайтный барьер. 15 марта компания стала публичной – в продажу поступили акции *Oracle Corporation*. Ларри Эллисон с состоянием в 185 млн долларов США попал в рейтинг журнала *Forbes*.

В 1988 году выпущена версия *Oracle* 6, в которой были представлены низкоуровневая блокировка, генерация последовательностей, отложенные записи, а

также другие изменения, увеличившие производительность и функциональность. Также появилась поддержка встроенного языка *PL/SQL* в средстве разработки приложений *Oracle Forms v*3 (в 6-й версии СУБД не было поддержки *PL/SQL*), а чуть позже объявлено, что *Oracle* 6.2 будет поддерживать симметричные кластеры с применением *Oracle Parallel Server*.

В 1989 году *Oracle* реализовала поддержку систем обработки транзакций в реальном времени (*OLTP – Online Transaction Processing*), продукты фирмы стали доступны в 86 странах мира. На выставке «Автоматизация-89» осенью появился стенд *Oracle* и примерно год спустя в СССР поступили в продажу лицензионные версии СУБД *Oracle*. Компания перенесла свою штаб-квартиру в Рэдвуд Шорз, штат Калифорния.

В 1990 году *Oracle* столкнулась с кризисом – ее стоимость неожиданно упала на 80 %. Эта отчаянная ситуация могла привести компанию к банкротству, и, дабы этого избежать, Эллисон пошел на крайние меры по увольнению сотрудников (стратегия «дело важнее людей»).

В 1992 году выпущена версия *Oracle* 7 для *Unix*. Эта СУБД поддерживала распределенные запросы, удаленное администрирование, осуществляла поддержку различных сетевых протоколов, поддержку ссылочной целостности, хранимых процедур и триггеров. Были введены такие новшества, как зеркальный журнал транзакций, динамическое создание журналов данных, команда ANALYZE, которая позволяла собирать статистику об использовании таблиц, индексов и других физических объектов, профили пользователей на использование системных ресурсов (ЦПУ, ввода/вывода и т. д.). Также в данной версии были полностью реализованы декларативные ограничения референциальной целостности в соответствии со стандартами *ANSI*/*ISO*.

В 1994 году выпущена версия *Oracle* 7 для IBM PC (до этого времени компания *Oracle* не рассматривала данную платформу как серверную, ограничиваясь лишь созданием для нее клиентских частей своих СУБД). В данной версии достигнута полная интеграция *PL/SQL* и *SQL*, введен стандартный пакет DBMS_SQL и асинхронная симметричная репликация данных вместе с асинхронным вызовом удаленных процедур.

В 1997 году выпущена версия *Oracle* 8, основными особенностями которой стали более высокая надежность по сравнению с предыдущей версией, а также поддержка большего числа пользователей и больших объемов данных. Выпущены версии *Oracle Designer*/2000 2.1 и *Oracle Developer*/2000, которые существенно облегчили создание приложений, работающих с БД, а также *Oracle Applications for Web*. Появились объектные типы и партиционирование таблиц.

В 1999 году выпущена версия *Oracle* 8*i* (*Oracle* 8.1), которая во многом была основана на применении *Java*. В данном названии буква «*i*» обозначала *Internet,* оповещая пользователя о том, что версия обладает поддержкой интернета. Виртуальная *Java*-машина находится в самой СУБД, на этом же языке написаны клиентские утилиты, инсталлятор, средства администрирования. В конце того же года выпущена вторая версия *Oracle* 8*i* (*Oracle* 8.1.6), поддерживающая *XML* и содержащая некоторые новшества, связанные с созданием хранилищ данных.

В 2001 году вышла версия *Oracle* 9*i* Release 1 (9.0.1), в которой было сделано более 400 изменений по сравнению с предыдущей. Среди них – «интеллектуализация» автоматизированных систем и расширение возможностей для аналитики. Появились средства обработки *XML*-документов; технология *Oracle RAC* (*Real Application Clusters*), механизм создания репликаций *Oracle Streams*; встроенная в СУБД поддержка интерактивной аналитической обработки (*OLAP – online analytical processing*) и *Data Mining*; переименование столбцов и ограничений целостности; поддержка *Java* 1.3.1 и *Unicode* 3.1.

В 2004 году появилась версия *Oracle* 10*g Release* 1 (10.1.0). Буква «*g*» в названии обозначает *Grid* («сеть») и символизирует поддержку распределенных *Grid*-вычислений. Год стал одним из самых успешных в истории компании – норма прибыли составила 38 % (самый высокий показатель за все время существования корпорации). Офис *Oracle* в России и СНГ вошел в тройку лучших представительств *Oracle* по темпам роста в регионе *EMEA* (Европа, Ближний Восток и Африка), а также пятый год подряд – в пятерку лучших среди 145 представительств *Oracle* в мире. В конце года *Oracle* приобрела *PeopleSoft* за 10,3 млрд долларов США.

В 2005 году анонсирована *Oracle* 10*g Release* 2 (10.2.0.1).

В 2006 году компания объявила о своем желании создать собственную операционную систему на базе *Linux*.

В 2007 году появилась версия *Oracle* 11*g Release* 1 (11.1.0.6).

В 2008 году корпорация выпустила на основе серверов *ProLiant* компании *Hewlett-Packard* и собственных решений для СУБД аппаратно-программный комплекс *Exadata*, заявив о своих амбициях на серверном рынке.

В 2009 году компания выпустила *Oracle* 11*g Release* 2 (11.2.0.1). В версию была введена новая для *Oracle* возможность «горячего» (без остановки сервера) внесения изменений в метаданные и бизнес-логику на *PL/SQL* – это стало возможным благодаря механизму одновременной поддержки нескольких версий схемы и логики под названием *editions*.

В 2010 году компания выпустила аппаратное обеспечение (для серверов *Sun*). 27 января произошло слияние *Sun Microsystems* и *Oracle*, несмотря на то что еврокомиссия некоторое время не могла одобрить данный союз, так как не хотела, чтобы под контролем *Oracle* оказалась разработка свободной СУБД *MySQL*. Став поставщиком серверного аппаратного обеспечения благодаря поглощению *Sun*, компания существенно расширила и линейку аппаратно-программных комплексов. Так, корпорация выпустила *Exalogic* (предконфигурированный кластер серверов приложений на основе оборудования *Sun* и программного обеспечения *WebLogic*), комплекс *Exalytics* (сервер аналитической обработки в оперативной памяти) и комплекс *Big Data Appliance* (*NoSQL*-кластер под управлением *Hadoop* и *R* в рамках реализации концепции «больших данных»). С начала 2010-х годов компания включилась в соперничество на рынке облачных технологий, позиционируя свои решения для частных облаков и открыв собственное публичное облако, предоставляющее как технологическое программное обеспечение по модели *PaaS*, так и бизнес-приложения по модели *SaaS*.

В 2011 и 2012 годы поглощены такие крупные производители облачных приложений, как *Taleo* и *RightNow*.

В 2013 году вышла версия 12*c* (12.1.0.1), основным новшеством которой была поддержка подключаемых БД (*pluggable database*), обеспечивающая свойства мультиарендности и живой миграции БД, суффикс «*c*» в названии обозначает *cloud* («облако»). 24 апреля 2015 года стало известно о планах *Oracle* перевести почти все свои продукты в облако. Таким образом, американская компания решила изменить свою бизнес-модель, чтобы соответствовать изменениям на рынке.

В 2016 году Ларри Эллисон объявил о создании в *Oracle* дата-центров для работы с *IaaS* второго поколения и заявил, что лидерство компании *Amazon* на облачном рынке подходит к концу. Цель компании – предложить клиентам *Oracle* пакет услуг, где будут совмещены *IaaS, PaaS* и *SaaS*.

В 2018 году вышла версия *Oracle* 18*c* (12.2.0.2). В конце 2017 года корпорация *Oracle* представила первую в мире автономную облачную БД, и уже в июне 2018 года она стала доступна. Одними из главных преимуществ БД являются способность системы самостоятельно, без приостановки работы обеспечивать защиту от киберугроз, а также отсутствие необходимости в ручном труде для администрирования базы. Произошли существенные изменения в процессе внедрения патчей в *Oracle Database* и *Grid Infrastructure*. За счет автоматической оптимизации кэширования, индексирования, параллельного выполнения, распределения и сжатия данных автономная БД будет менее требовательной к системным ресурсам.

В 2019 и 2020 годы выходят новые версии *Oracle Database* 19*c* и 20*c*, их возможности иллюстрируют перспективные направления развития современных средств работы с БД:

– создание мощных коммерческих СУБД, одинаково эффективно работающих в разных средах (ЦОД заказчика, частное, публичное или гибридное облако, машина баз данных *Exadata, Oracle Cloud@Customer* – фрагмент публичного облака, развернутый в ЦОД заказчика). Иначе говоря, современная база данных может работать и как СУБД, и как сервис;

– самоуправляемость, автономность и одновременное управление множеством БД с использованием алгоритмов искусственного интеллекта;

– мультимодельность и конвергентность – поддержка в одной СУБД множества типов данных (реляционные, геоинформация, текст, *JSON, NOSQL, XML, Hadoop, Spark*, мультимедиа и пр.) и множества разных нагрузок (*OLTP*, системы поддержки принятия решений (СППР), интернет вещей, хранилища данных, системы блокчейн, «ключ – значение» и т. д.);

– вычисления *In-memory* и использование векторных команд процессоров;

– работа с энергонезависимой памятью;

– разделение БД между несколькими серверами (шардинг);

– встраивание новых технологий в СУБД (блокчейн, машинное обучение, интернет вещей, *JSON* и т. д.).

Кроме того, продолжается работа над повышением надежности, производительности, безопасности, масштабируемости и управляемости СУБД, каждый год *Oracle* обновляет линейку продуктов и сохраняет год в названии версии. Материал подраздела основан на информации из источника [1].

1.2. Основные продукты компании и редакции СУБД *Oracle*

Выделяют три группы продуктов в семействе технологий *Oracle*: сервер БД, сервер приложений и систему управления. Это основные компоненты для организации сетевых вычислений. Основной концепцией для построения инфраструктуры выступает виртуализация. Пользователи работают с информацией (обычно посредством веб-приложения), но они не знают и не должны знать, откуда и как эти данные выводятся. Упрощенно БД отвечает за хранение и выдачу информации, сервер приложений – за инфраструктуру и развертывание служб нужных пользователю, а система управления используется для администрирования и управления. Платформа или физические серверы, используемые для работы, не имеют значения для конечного пользователя. Виртуализация позволяет эффективно распределять ресурсы, предоставляя максимальную производительность там, где это нужно (балансировка нагрузки) [2].

Сервер БД *Oracle* включает в себя экземпляр (*instance*) БД и саму БД с множеством возможностей, таких как потоки, партиционирование, хранилище данных, репликация и *RAC* (*Real Application Cluster*), но самое главное – это надежное, высокопроизводительное хранилище данных, построенное на объектно-ориентированной системе для БД. *Oracle Server* – это основной продукт *Oracle*, который продолжает активно развиваться и является ядром других продуктов.

БД – это набор файлов в дисковой системе. База существует, пока существуют файлы. Теоретически нет ограничений по размеру и количеству файлов, таким образом, нет ограничений размера БД. Доступ к данным происходит через экземпляр (*instance*) сервера БД. Инстанс – набор процессов и структур данных в памяти. Он может быть запущен и остановлен. Пользователи подключаются к инстансу и уже он управляет доступом к данным. Невозможно работать с данными напрямую. Инстанс и файлы БД вместе и создают сервер *Oracle*.

Такая модель доступа является клиент-серверной моделью, также известной как двухуровневая (*two-tier*) модель. В клиент-серверной модели пользовательский интерфейс и логика приложения не зависят от управления данными. Для приложения, разработанного с использованием *SQL*, это значит, что пользовательская часть приложения создает *SQL*-запросы, а серверная часть исполняет их. Это классическое разделение клиентской и серверной части, обычно соединенной локальной сетью. Сетевой протокол, используемый в *Oracle*, закрытый и называется *Oracle Net*.

Клиентская часть состоит из двух компонентов: пользователей и пользовательских процессов. В серверной части три составляющие: серверный процесс, который исполняет *SQL*-запросы, инстанс (*instance*) и сама БД. Каждый пользователь взаимодействует с пользовательским процессом, который в свою очередь

взаимодействует с серверным процессом, обычно посредством локальной сети. Серверный процесс взаимодействует с экземпляром, и экземпляр – с БД.

Сессия – это пользовательский процесс с серверным процессом. Обычно это один пользовательский процесс для пользователя и серверный процесс для каждого пользовательского процесса. Сессия, как правило, создается по запросу пользователя и уничтожается, когда она больше не нужна: это называется *log-on* и *log-off* цикл. Экземпляр и структуры в памяти, необходимые для работы, запускаются администратором и существуют, пока администратор не остановит их: это называется запуск и остановка экземпляра сервера БД.

Пользовательским процессом может выступать любое клиент-серверное приложение, которое может подключиться к *Oracle*-серверу. В частности, рекомендуется использовать следующее программное обеспечение, предоставляемое *Oracle* для подключения к серверу и выполнения *SQL*-запросов: *SQL*Plus* (программа-интерпретатор командной строки для работы с СУБД *Oracle Database*; как правило, устанавливается вместе с БД) и *SQL Developer* (интегрированная среда разработки на языках *SQL* и *PL/SQL* с возможностью администрирования БД).

Сервер приложений. Со становлением веб-приложений как стандарта для работы пользователей появилась потребность в серверах приложений, которые позволяют заменить установленное на компьютере пользователя приложение на приложения, установленные в определенном хранилище. Пользовательский интерфейс отображается посредством браузера. Такие приложения могут использовать данные, находящиеся в разных БД. Сервер приложений *Oracle* – это платформа для разработки, установки и управления веб-приложениями. Веб-приложение – любая программа, работающая с использованием протокола *HTTP*. Веб-приложения, как правило, используют трехуровневую (*three-tier's*) модель: уровень БД для доступа к данным, пользовательский уровень (им обычно выступает веб-браузер), цель которого отображение окон и диалогов для взаимодействия с пользователем, и уровень приложения между ними, который реализует бизнес-логику для генерации пользовательского интерфейса и выполнения запросов к БД.

Возможно использовать отдельное для каждого пользователя соединение: каждый пользователь будет создавать сессию к серверу приложений, который в свою очередь будет создавать подключение к БД. Однако эта модель признана очень неэффективной по сравнению с пулами подключений (*connection pooling model*). Согласно модели пула подключений сервер приложений создает относительно небольшое количество соединений с БД и делает их доступными для запросов (помещая при необходимости запросы в очередь) относительно большого количества пользователей сервера приложений. С точки зрения БД нет абсолютно никакой разницы между запросами из *SQL*Plus* или пула запросов. В первом случае пользовательский процесс происходит на одной машине, во втором случае пользовательский процесс разбит на два уровня: сервер приложений генерирует пользовательский интерфейс и пользовательский уровень отображает его.

Система управления. Увеличение объема и сложности *IT*-установок может сделать управление компонентами достаточно сложной процедурой. Инструменты управления могут сделать эту задачу легче и значительно повысить производительность работы. *Oracle Enterprise Manager* включает в себя три группы инструментов:

– управление БД (*Database Control*);

– управление сервером приложений (*Application Server Control*);

– управление инфраструктурой (*Oracle Enterprise Manager Grid Control*).

Управление БД – графический инструмент для управления одной БД, которая может быть *RAC*-кластером. Имеется возможность управления и мониторинга в режиме реального времени, планировщик задач и резервного копирования, генерация отчетности.

Управление сервером приложений – это инструмент для управления несколькими серверами приложений. Технология управления зависит от версии. До *Oracle*10*gR*2 включительно используется технология «ферма» с репозиторием метаданных и центральным управляющим входом. Эта модель закрытая и предоставляет отличные возможности для установки и поддержи приложений. Начиная с версии *Oracle*10g*R*3 используется технология, основанная на *J2EE* (*Java™ 2 Platform, Enterprise Edition*) кластеризации.

И *Database Control*, и *Application Server Control* состоят из *java*-процессов, запущенных на сервере, которые ожидают *HTTP*- или *HTTPS*-подключений. Администраторы подключаются к ним с помощью браузера. *Database Control* подключается к БД, а *Application Server Control* – к серверу приложений.

Инструмент *Oracle Enterprise Manager Grid Control* обобщает управление инфраструктурой. Репозиторий (находящийся внутри БД *Oracle*) и один или несколько серверов управляют всей средой: всеми БД и серверами приложений, расположенными физически абсолютно в любой точке мира. Данный инструмент также может управлять узлами или машинами, на которых запущены серверы, и с помощью плагинов различными сторонними продуктами. Каждый управляемый узел запускает процесс, который ответственен за мониторинг управляемых процессов на узле: этот процесс выполняет разные задачи и возвращает результаты на сервер управления.

Инструмент *Oracle Enterprise Manager Grid Control* дает целостное представление обо всей инфраструктуре и может кардинально улучшать производительность администраторов. С его помощью один администратор может обслуживать десятки и сотни серверов.

Инструменты разработчика *Oracle*. *Oracle* предоставляет различные инструменты для разработки программ и утилит и поддерживает различные языки программирования. Языки программирования, которые разбираются на инструкции и выполняются внутри *Oracle* сервером, – это *SQL, PL/SQL* и *Java*. Технологии для разработки вне БД можно найти в *Oracle Developer Suite* (*Forms, Reports and Discoverer*), *Oracle Application Server* и других языках третьего уровня

(3*GLs*). Также доступен широкий выбор инструментов, которые могут использоваться для подключения к БД *Oracle*, например *Microsoft.NET*. Кроме того, для обучения, начиная с версии 11*g*, можно не устанавливать сервер СУБД *Oracle*, а использовать *Oracle Application Express* (*Oracle Apex*) – проприетарную среду быстрой разработки прикладного программного обеспечения на основе СУБД *Oracle Database*, целиком реализованную как веб-приложение и предоставляющую возможность получить до 12 Гбайт рабочего пространства на облачном сервере *Oracle* в редакции *XE* [3].

Встроенные языки. *SQL* используется для работы с данными, но на нем нельзя создать полноценное приложение, так как нет возможности для создания пользовательского интерфейса и отсутствует поддержка сложных структур данных. Два других внутренних языка программирования устраняют эти пробелы. Это *PL/SQL* и *Java*. *PL/SQL* – язык третьего уровня и является собственностью *Oracle*. Он поддерживает стандартные управляющие конструкции: алгоритмы ветвления (IF, THEN, ELSE) и циклы, имеет возможность поддержки сложных структур данных. *SQL*-запросы могут быть частью *PL/SQL*-кода, таким образом программа *PL/SQL* может использовать *SQL* для получения данных из БД, выполнять определенные действия в зависимости от данных и затем выполнять другие запросы для записи данных назад в базу, что обеспечивает взаимодействие пользователей с приложением. Кроме того, *Java* может выполнять *SQL*-запросы, написанные внутри *Java*-кода, а также восполняет пробелы с созданием пользовательского интерфейса. Это стандарт технологии: любой *Java*-программист должен быть способен написать код, который будет работать в БД *Oracle* (либо другой *Java*-совместимой БД).

Все администраторы БД должны хорошо знать *SQL* и *PL/SQL*. Это стандартное и необходимое требование. Знание *Java* уже не особо обязательно, так как *Java* редко используется при работе непосредственно с БД. Раньше *Oracle Application Server* не мог запускать некоторые стандартные компоненты (например, *Servlet* и *EJB* (*Enterprise JavaBeans*)). Чтобы устранить это, *Oracle* разработали *Java*-машину, встроенную в БД, которая соответствует стандартам и, начиная с *Oracle Application Server* 9*i*, стало возможным запускать *J2EE*-компоненты там, где им и положено быть: на сервере приложений. Благодаря этому и стало запускаться меньше *Java*-кода внутри БД.

Обычно *DBA* (*database administrator*) проводят много времени над задачами повышения производительности и отладки *SQL*- и *PL/SQL*-кода. С точки зрения *Oracle* администратор должен находить проблемные участки и передавать разработчику для исправления, однако на практике разработчикам не хватает знания (или желания) делать это и администраторам приходится брать на себя эту роль.

Основные редакции СУБД *Oracle* [1]. СУБД *Oracle* поставляется в пяти основных локальных (*On-Premise*) редакциях, ориентированных на различные

сценарии разработки и развертывания приложений, а также отличающихся ценой. Перечень редакций приведен табл. 1.1.

Таблица 1.1

Перечень редакций СУБД *Oracle*

Название	Описание
Oracle Database Enterprise Edition (EE)	Корпоративная редакция, которая является самой дорогой и функциональной, предоставляет наибольшую производительность, доступность, масштабируемость и безопасность, критичные для бизнес-приложений. Используется для разработки *OLTP*-приложений с целью обработки больших объемов транзакций, хранилищ данных с интенсивными запросами и требовательных интернет-приложений. *Oracle Database Enterprise Edition* содержит все компоненты *Oracle Database*, но может быть дополнительно расширена путем приобретения дополнительных опций и пакетов
Oracle Database Standard Edition 2 (*SE*2)	Стандартная редакция, включающая в себя все средства, необходимые для создания критически важных бизнес-приложений, не поддерживает кластеризацию *Oracle Real Application Clusters*
Oracle Database Enterprise Edition on Engineered Systems (*EE-ES*)	Специальная редакция для установки в локальной системе *Oracle Exadata Database Machine* или *Oracle Database Appliance*, включает в себя все компоненты *Oracle Database*, но может быть дополнительно расширена путем приобретения опций и пакетов
Oracle Database Personal Edition (*PE*)	Редакция поддерживает однопользовательские среды разработки и развертывания, требующие полной совместимости с *Oracle Database Standard Edition* 2 и *Oracle Database Enterprise Edition*. Данная редакция включает в себя все компоненты и опции, входящие в *Enterprise Edition*, за исключением опции *Oracle RAC One Node* и *Oracle Real Application Clusters*
Lite	Редакция для мобильных и встраиваемых устройств
Oracle Database Express Edition (*XE*)	Бесплатная редакция, которую можно использовать для обучения или для разработки небольших приложений В *Oracle* 18*c* используется два процессора, оперативная память – 2 Гбайта, максимальный объем БД – 12 Гбайт

Кроме редакций, которые можно установить локально, есть еще и несколько облачных (*cloud*) редакций:

1. *Oracle Database Cloud Service Standard Edition* (*DBCS SE*);
2. *Oracle Database Cloud Service Enterprise Edition* (*DBCS EE*);
3. *Oracle Database Cloud Service Enterprise Edition – High Performance* (*DBCS EE-HP*);
4. *Oracle Database Cloud Service Enterprise Edition – Extreme Performance* (*DBCS EE-EP*);
5. *Oracle Database Exadata Cloud Service* (*ExaCS*).

Также *Oracle* производит операционные системы и платформы виртуализации: *Oracle VM* – решение для серверной виртуализации на основе гипервизора *Xen* на платформах *x*86; *Oracle VM Server for SPARC* – решение, основанное на функции логических доменов, позволяющее запускать до 128 виртуальных машин на серверах под управлением процессоров *SPARC T*-серии; *Oracle VM VirtualBox* – система персональной виртуализации *x*86. От *Sun Microsystems* унаследованы операционные системы семейства *Solaris* и проект *OpenSolaris*.

В 2006 году *Oracle* выпустила клон дистрибутива *Red Hat Enterprise Linux* под наименованием *Oracle Enterprise Linux* (*Oracle Linux*) и с этого момента оказывает коммерческую и юридическую поддержку пользователям данного дистрибутива. Несмотря на намеренно несущественные отличия от дистрибутива *Red Hat*, многие поставщики серверного оборудования, в том числе *Dell*, *Hewlett-Packard*, *IBM*, поддержали *Oracle Linux*. Основные отличия по использованию – это вдвое более низкая стоимость подписки на сопровождение и необязательность ее приобретения для получения обновлений.

Комплект бизнес-приложений собственной разработки, начатой в конце 1980-х годов под названием *Oracle Applications*, в 2000-е был переименован в *E-Business Suite* и реализует функциональные возможности *ERP*, *CRM*, *SCM*, *EAM*. В результате поглощения *PeopleSoft* в этой продуктовой линейке *Oracle* появились следующие универсальные бизнес-приложения: *Peoplesoft Enterprise*; *JD Edwards EnterpriseOne*; *JD Edwards OneWorld*.

Также в продуктовом портфеле корпорации появилось большое число проблемно-ориентированных бизнес-приложений, среди которых выделяются:

1. *Siebel CRM* – система-лидер *CRM*-рынка (*customer relationship management*) на момент поглощения *Siebel* в 2005 году;
2. *Primavera* – ориентированное на крупных заказчиков приложение по управлению проектами;
3. *Hyperion*, также выпускаемый как *Oracle EPM*, – система класса *EPM* (*Enterprise Project Management*), обеспечивающая финансовое планирование, управление показателями эффективности, финансовый анализ для корпоративных заказчиков;
4. *Agile*, *PLM*-система (*Product Lifecycle Management*).

Партнерская сеть в Республике Беларусь в досанкционный период включала более 30 компаний и работала с 1991 года, имела центр технической поддержки, центр авторизованного обучения и три центра компетенции, с 2006 по 2020 год в Минске практически ежегодно проводились конференции мирового уровня *Oracle Day*.

Основные группы заказчиков в Республике Беларусь – это государственный сектор (министерства, ведомства, институты, комитеты), финансовый сектор (банки, биржи), телекоммуникации, производственные предприятия, нефтяные, газовые, транспортные компании, организации страхования, розничной торговли, здравоохранения и т. д.

1.3. Общие сведения о *SQL*

Язык структурированных запросов *SQL* (*Structured Query Language*) является декларативным языком четвертого поколения, предназначенным для работы с реляционными базами данных.

В начале 1970-х годов в одной из исследовательских лабораторий компании *IBM* была разработана экспериментальная реляционная СУБД *System R*, для которой затем был создан специальный язык *SEQUEL*, позволявший относительно просто управлять данными в этой СУБД. Аббревиатура *SEQUEL* расшифровывалась как *Structured English QUEry Language* – структурированный английский язык запросов. Позже по юридическим соображениям язык *SEQUEL* был переименован в *SQL* и официальным произношением стало [*‚es kju:' el*] – эс-кью-эл.

Первой СУБД, поддерживающей новый язык, стала в 1979 году *Oracle* 2.

Первый официальный стандарт языка *SQL* был принят *ANSI* (*American National Standards Institute*) в 1986 году и *ISO* (*International Organization for Standardization*) в 1987 году (так называемый *SQL*-86) и несколько уточнен в 1989 году. Дальнейшее развитие языка поставщиками СУБД потребовало принятия в 1992 году нового расширенного стандарта (*ANSI SQL*-92 или просто *SQL*2). Следующим стандартом стал *SQL*:1999 (*SQL*3). Долгое время действовал стандарт, принятый в 2003 году (*SQL*:2003) с небольшими модификациями, внесенными позже. Последняя, девятая версия стандарта была принята в июне 2023 года.

История версий стандарта представлена в табл. 1.2.

Таблица 1.2

История версий стандарта *SQL*

Год	Название	Синоним	Изменения
1986	*SQL*-86	*SQL*-87	Первый вариант стандарта, принятый институтом *ANSI* и одобренный *ISO* в 1987 году
1989	*SQL*-89	*FIPS* 127-1	Немного доработанный вариант предыдущего стандарта
1992	*SQL*-92	*SQL*2, *FIPS* 127-2	Значительные изменения (*ISO* 9075); уровень *Entry Level* стандарта *SQL*-92 был принят как стандарт *FIPS* 127-2

Окончание табл. 1.2

Год	Название	Синоним	Изменения
1999	*SQL*:1999	*SQL*3	Добавлены поддержка регулярных выражений, рекурсивных запросов, поддержка триггеров, базовые процедурные расширения, нескалярные типы данных и некоторые объектно-ориентированные возможности
2003	*SQL*:2003	–	Введены расширения для работы с *XML*-данными, оконные функции (применяемые для работы с *OLAP*-базами данных), генераторы последовательностей и основанные на них типы данных
2006	*SQL*:2006	–	Функциональность работы с *XML*-данными значительно расширена. Появилась возможность совместно использовать в запросах *SQL* и *XQuery*
2008	*SQL*:2008	–	Улучшены возможности оконных функций, устранены некоторые неоднозначности стандарта *SQL*:2003
2011	*SQL*:2011	–	Реализована поддержка хронологических баз данных (PERIOD FOR), поддержка конструкции FETCH
2016	*SQL*:2016	–	Защита на уровне строк, полиморфные табличные функции, *JSON*
2023	*SQL*:2023	–	Добавлено расширение *SQL/PGQ* (*Property Graph Queries*) для манипуляции наборами связанных между собой данных, образующих граф, значительно расширены возможности, связанные с обработкой данных в формате *JSON*

Перечислим основные преимущества использования стандарта *SQL*.

1. Независимость от конкретной СУБД. Несмотря на наличие диалектов и различий в синтаксисе, в большинстве своем тексты *SQL*-запросов, содержащие *DDL*- и *DML*-операторы, могут быть достаточно легко перенесены из одной СУБД в другую. Естественно, что при применении некоторых специфичных для реализации возможностей такой переносимости добиться очень трудно.

2. Наличие стандартов. Наличие стандартов и набора тестов для выявления совместимости и соответствия конкретной реализации *SQL* общепринятому стандарту способствует «стабилизации» языка. Однако стандарт местами чересчур формализован и «раздут» в размерах, например основная часть стандарта *SQL*:2003 представляет собой более 1300 страниц текста.

3. Декларативность. С помощью *SQL* программист описывает, какие данные нужно извлечь или модифицировать. То, каким образом это сделать, решает СУБД непосредственно при обработке *SQL*-запроса, но при этом полезно представлять, как СУБД будет разбирать текст его запроса. Особенно критичными такие моменты становятся при работе с большими БД и со сложными запросами, – чем сложнее сконструирован запрос, тем больше он допускает вариантов написания, различных по скорости выполнения, но одинаковых по итоговому набору данных.

Кроме этого, можно отметить и ряд недостатков:

1. Несоответствие реляционной модели данных. Создатели реляционной модели данных Эдгар Кодд, Кристофер Дейт и их сторонники указывают на то, что *SQL* не является истинно реляционным языком. В частности, они указывают на следующие проблемы *SQL*:

– повторяющиеся строки;

– неопределенные значения (NULL);

– явное указание порядка колонок слева направо;

– колонки без имени и дублирующиеся имена колонок;

– использование указателей и т. д.

2. Сложность. Хотя *SQL* и задумывался как средство работы конечного пользователя, в конце концов он стал настолько сложным, что превратился в инструмент программиста.

3. Отступления от стандартов. Несмотря на наличие международного стандарта *ANSI SQL*-92, многие компании, занимающиеся разработкой СУБД (например, *Oracle, Sybase, Microsoft, MySQL*), вносили и вносят изменения в язык *SQL*, применяемый в разрабатываемой СУБД, тем самым отступая от стандарта. Таким образом, появляются специфичные для каждой конкретной СУБД диалекты языка *SQL*. Существует четыре уровня соответствия реализации *SQL* стандарту:

– *entry* (базовый);

– *transitional* (переходный);

– *intermediate* (промежуточный);

– *full* (полный).

4. Сложность работы с иерархическими структурами. Ранее *SQL* не предлагал стандартного способа манипуляции древовидными структурами. Некоторые поставщики СУБД предлагали свои решения. Например, *Oracle* использует выражение CONNECT BY. В настоящее время в качестве стандарта работы с иерархическими данными принята рекурсивная конструкция WITH.

Поскольку *SQL* не является языком программирования (т. е. не предоставляет средств для автоматизации операций с данными), вводимые разными производителями расширения касались в первую очередь процедурных расширений. Это хранимые процедуры и процедурные языки. Практически в каждой СУБД применяется свой процедурный язык. Подобные языки для самых популярных СУБД приведены в табл. 1.3.

Таблица 1.3

Процедурные языки различных СУБД

СУБД	Краткое название	Расшифровка
Borland InterBase, Firebird	*PSQL*	*Procedural SQL*
IBM DB2	*SQL PL*	*SQL Procedural Language* (расширяет *SQL/PSM*)
Microsoft SQL Server, Sybase ASE	*Transact-SQL*	*Transact-SQL*
MySQL	*SQL/PSM*	*SQL/Persistent Stored Module* (стандарт *SQL:2003*)
Oracle	*PL/SQL*	*Procedural Language/SQL* (основан на языке *Ada*)
PostgreSQL	*PL/pgSQL*	*Procedural Language/PostgreSQL* (очень похож *на Oracle PL/SQL*)

Материал подраздела подготовлен на основе информации в источнике [4].

1.4. Основные правила написания *SQL*-операторов

В *SQL*-операторах необходимо придерживаться точных синтаксических и структурных правил, правильно использовать ключевые слова, а также идентификаторы, операторы и константы. Символы, используемые в различных частях *SQL*-оператора, могут зависеть от реализации СУБД и определяются в основном множеством символов, которое *SQL*-сервер использует по умолчанию.

SQL-операторы часто называют инструкциями. Инструкция – это набор ключевых слов, связанных друг с другом и используемых как единое целое в командах *SQL*. В составе *SQL* условно выделяют следующие группы инструкций:

– *DDL* (*Data Definition Language*) – инструкции языка описания данных;

– *DML* (*Data Manipulation Language*) – инструкции языка манипулирования данными;

– *DCL* (*Data Control Language*) – инструкции языка для предоставления и отзыва привилегий пользователей БД;

– *TCL* (*Transaction Control Language*) – инструкции языка для обеспечения согласованности БД и управления транзакциями.

К общим и наиболее распространенным инструкциям *SQL* относятся:

– инструкция_авторасширения (AUTOEXTEND), которая указывает, разрешено ли для файла увеличение его размера;

– инструкция_ограничения_столбца (таблицы) (CONSTRAINT), которая задает ограничение целостности столбца (таблицы);

– инструкция_состояния_ограничения (DEFERRABLE), которая позволяет избирательно применять и не применять ограничение целостности;

– инструкция_атрибутов_индекса, которая включает инструкцию_физических_атрибутов и инструкцию_хранения и задает физические и логические характеристики индекса.

Каждая инструкция *SQL* начинается с ключевого слова, описывающего действие, выполняемое инструкцией (например, CREATE, INSERT, DELETE, COMMIT), после чего следует одно или несколько предложений. Предложение описывает данные, с которыми работает инструкция, или содержит уточняющую информацию о действии, выполняемом инструкцией. Каждое предложение также начинается с ключевого слова, например, такого как WHERE, FROM, INTO, HAVING. Одни предложения в инструкции являются обязательными, другие – нет. Конкретная структура и содержимое предложения могут изменяться. Многие предложения содержат имена таблиц или столбцов; некоторые из них могут содержать дополнительные ключевые слова, константы и выражения. В соответствии со стандартом ключевые слова нельзя использовать для именования объектов БД, таких как таблицы или столбцы.

При написании *SQL*-операторов следует учитывать следующее:

– *SQL*-операторы не чувствительны к регистру за исключением символьных данных в полях таблиц;

– запись *SQL*-оператора имеет свободный формат, однако существует ряд общепринятых рекомендаций, которые облегчают читабельность *SQL*-программ:

– ключевые (зарезервированные) слова следует писать прописными буквами;

– слова, определяемые пользователем, следует писать строчными буквами;

– ключевые слова в пределах одного *SQL*-оператора следует писать с новой строки и выравнивать по одной и той же позиции. При использовании вложенных операторов SELECT их следует сдвигать внутрь предложения на позицию второго слова в предыдущей строке;

– если фраза *SQL*-оператора превышает длину строки, то ее продолжение следует смещать относительно начала следующей строки.

В описании синтаксиса *SQL*-операторов используются следующие специальные символы:

– вертикальная черта (|), которая указывает на необходимость выбора одного из нескольких значений (например, *a* | *b* | *c*);

– фигурные скобки ({}), которые указывают, что заключенный в них элемент является обязательным;

– квадратные скобки ([ ]), которые указывают, что заключенный в них элемент является необязательным;

– многоточие (...), которое используется для указания необязательной возможности повторения указанной конструкции от нуля до нескольких раз (например, запись {*a* | *b*}, [*c* …] означает, что за *a* или *b* может следовать от нуля до нескольких повторений *c*, разделенных запятыми.

– звездочка (*) для обозначения «все» – употребляется в обычном для программирования смысле, т. е. «все случаи, удовлетворяющие определению»;

– точка с запятой (;) – завершающий элемент предложений *SQL*;

– запятая (,) – используется для разделения элементов списков;

– пробелы ( ) – могут вводиться для повышения наглядности между любыми синтаксическими конструкциями предложений *SQL*.

Имена (идентификаторы) таблиц и столбцов. Имена используются в инструкциях *SQL* для указания, над каким объектом БД инструкция должна выполнить действие. Основными именованными объектами в реляционной БД являются таблицы, столбцы, пользователи, схемы, ограничения, домены и ряд других объектов.

Стандарт *SQL* задает набор символов, который используется по умолчанию, он включает строчные и прописные буквы латинского алфавита (*A–Z, a–z)*, цифры (0–9) и символ подчеркивания (_). На формат идентификатора накладываются следующие ограничения:

– имеет длину до 128 символов;

– начинается с буквы;

– не может содержать пробелы.

На практике поддержка имен в различных СУБД реализована по-разному, поэтому для повышения переносимости лучше делать имена сравнительно короткими и избегать употребления в них специальных символов.

Если в инструкции *SQL* указано имя таблицы, то СУБД по умолчанию обращается к одной из собственных таблиц пользователя. Обычно таблицам присваиваются короткие и значимые имена. В небольших БД, предназначенных для личного или группового использования, выбор имен зависит от разработчика БД. В более крупных, корпоративных БД могут существовать определенные корпоративные стандарты именования таблиц, позволяющие избежать конфликтов имен.

Кроме этого, пользователь может создавать именованное множество таблиц, называемое схемой. При этом пользователь может создавать таблицы с одинаковыми именами в различных схемах. Обращаться к таблицам, которые имеют одинаковые имена, можно с помощью полного имени таблицы, т. е. имени, включающего идентификатор владельца или схемы и непосредственно имя таблицы, указанные через точку. Например, полное имя таблицы *branch*, владельцем которой является пользователь по имени *user*, имеет следующий вид: *user.branch*. При обращении к таблицам другого пользователя необходимо иметь на это соответствующее разрешение. Сразу следует отметить, что это в *Oracle* пользователь не создаст несколько схем одновременно, так как имя пользователя отождествляется с именем схемы и, таким образом, каждый пользователь владеет только одной схемой.

Если в инструкции задается имя столбца, СУБД сама определяет, в какой из указанных в этой же инструкции таблиц содержится данный столбец. Если в инструкцию требуется включить два столбца из различных таблиц, но с одинаковыми именами, необходимо указывать полные имена столбцов. Полное имя столбца состоит из имени таблицы, содержащей столбец, и непосредственно имени столбца, разделенных точкой.

Полные имена таблиц и столбцов можно использовать вместо коротких во всех инструкциях *SQL*.

Константы. В некоторых инструкциях *SQL* необходимо явно указывать значения чисел, строк или даты. В стандарте *ANSI*/*ISO* определен формат числовых и строковых констант, или литералов, которые представляют конкретные значения данных. Этот формат используется в большинстве СУБД.

Целые и десятичные константы (точные числовые литералы) в инструкциях *SQL* представляются в виде обычных десятичных чисел с необязательным знаком плюс или минус перед ними.

Константы с плавающей запятой (приблизительные числовые литералы) определяются с помощью символа *E* и имеют такой же формат, как и в большинстве языков программирования.

В соответствии со стандартом *ANSI/ISO* строковые константы (строковые литералы) в *SQL* должны быть заключены в одинарные кавычки.

Константы типа даты в диалекте *SQL* для *Oracle* имеют специфичный формат представления. Как и строковые литералы, даты при их явном представлении берутся в одинарные кавычки.

Кроме пользовательских констант, в *SQL* существуют специальные именованные константы, возвращающие значения, хранимые в самой СУБД. Так, например, в *Oracle* существует константа SYSDATE, при указании которой в запросе СУБД возвращает в результат или подставляет в поля БД текущую дату и время. Также стандартом регламентировано применение константы USER для фиксации имени пользователя, сделавшего изменения в записях таблиц БД и т. д.

Контрольные вопросы и задания к разделу 1

1. Какие основные редакции есть у СУБД *Oracle* и чем они различаются?
2. В каких сферах и почему используется СУБД *Oracle*?
3. Что означают буквы *i, c* и *g* в названиях версий СУБД *Oracle*?
4. Какие условные обозначения приняты в синтаксисе *SQL*?
5. Как правильно расшифровывается *SQL* и чем он отличается от других языков программирования?
6. Что представляет собой инструкция *SQL*?
7. Сколько символов может включать стандартный идентификатор *SQL*?
8. Какие положительные и отрицательные стороны есть у стандарта *SQL*?
9. Как расшифровываются аббревиатуры *DDL*, *DML*, *DCL*?
10. Назовите основные клиентские инструменты для работы с СУБД *Oracle*.
11. Создайте облачный аккаунт СУБД *Oracle Express Edition* по адресу https://apex.oracle.com/, войдите в него.
12. Изучите основное окно и настройки администрирования.
13. Ознакомьтесь с основными возможностями работы на вкладке *SQL Workshop*.
14. С помощью команд DESC и SELECT * FROM DICT ознакомьтесь со структурой и содержимым основного словаря данных DICT.

2. ОПЕРАТОРЫ ЯЗЫКА ОПРЕДЕЛЕНИЯ ДАННЫХ

2.1. Общие сведения об операторах *DDL*

С помощью инструкций *DDL* можно:
– определить и создать структуру новой таблицы;
– удалить таблицу, которая больше не нужна;
– изменить определение существующей таблицы;
– определить виртуальную таблицу (представление) данных;
– обеспечить безопасность БД;
– создать индекс для ускорения доступа к данным таблицы;
– управлять физическим размещением данных и т. д.

В большинстве случаев инструкции *DDL* обеспечивают высокий уровень доступа к данным и позволяют пользователю не вникать в детали хранения информации в БД на физическом уровне. Они оперируют абстрактными объектами БД, такими как таблицы и столбцы. Однако *DDL* не может совершенно не затрагивать вопросов, связанных с физической памятью. Инструкции и предложения *DDL*, управляющие физической памятью, могут быть разными в различных СУБД.

Стандартное ядро языка определения данных образуют три инструкции:
– CREATE (создать), позволяющая определять и создавать объекты БД, такие как, например, таблицы, индексы, представления, пользователи, роли и т. д.;
– DROP (удалить), служащая для удаления существующего объекта БД;
– ALTER (изменить), с помощью которой можно изменить определение уже существующего объекта БД.

Все основные реляционные СУБД позволяют использовать три указанные инструкции *DDL*. Таким образом, структура реляционной БД является динамической. Например, СУБД может создавать, удалять или изменять таблицы, одновременно с этим обеспечивая доступ пользователям к БД. Это одно из главных преимуществ реляционных БД по сравнению с более ранними системами, в которых изменять структуру БД можно было только после полного прекращения работы СУБД. Это означает, что с течением времени БД может расти и изменяться. Ее промышленная эксплуатация может продолжаться в то время, когда в БД добавляются все новые таблицы и подсхемы.

Хотя *DDL* и *DML* являются двумя отдельными частями *SQL*, в большинстве реляционных СУБД такое разделение существует лишь на абстрактном уровне. Обычно инструкции *DDL* и *DML* в СУБД абсолютно равноправны, и их можно произвольно чередовать как в интерактивном, так и в программном *SQL*. Если программе или пользователю требуется таблица для временного хранения результатов, они могут ее создать, заполнить, проделать с данными необходимую работу и затем удалить.

Список основных *DDL*-операторов, применяемых в *Oracle*, приведен в табл. 2.1.

Таблица 2.1

Список основных *DDL*-операторов

Оператор	Описание
ALTER RESOURCE COST	Изменяет формулу, по которой вычисляется общая стоимость ресурсов сеанса
ALTER SYSTEM	Производит динамическое изменение экземпляра БД
ASSOCIATE STATISTICS	Определяет, как будет вычисляться статистика для указанных объектов БД
CALL	Исполнение хранимого процесса или функции *PL/SQL*
COMMENT ON	Добавляет комментарий для таблицы, представления, моментальной копии или столбца в словаре данных
CREATE | ALTER | DROP Создает, изменяет или удаляет:	
CLUSTER	кластер – объект схемы, который содержит одну или несколько таблиц, имеющих несколько общих столбцов
FUNCTION	функцию
INDEX	индекс
JAVA	объект, содержащий исходный текст, класс или ресурс *Java*
MATERIALIZED VIEW	материализованное представление
OPERATOR	оператор
PACKAGE	пакет
PROCEDURE	процедуру
PROFILE	профиль
ROLE	роль
SNAPSHOT	моментальную копию
SEQUENCE	последовательность
TABLE	таблицу
TABLE SPACE	табличное пространство
TRIGGER	триггер
TYPE	тип
USER	пользователя
VIEW	представление
CREATE | DROP Создает или удаляет:	
CONTEXT	пространство имен для контекста
DATABASE LINK	ссылку на удаленную БД
DIMENSION	измерение
DIRECTORY	объект каталога
LIBRARY	объект схемы, сопоставленный разделяемой библиотеке ОС
INDEXTYPE	индексный тип
SYNONYM	общий или частный синоним

Окончание табл. 2.1

Оператор	Описание
CREATE Создает:	
CONTROLFILE	управляющий файл
TEMPORARY TABLESPACE	временное табличное пространство
SCHEMA	несколько таблиц и (или) представлений и выдает права на них в одной инструкции
CREATE | ALTER Создает или изменяет:	
DATABASE	базу данных
FLASHBACK	Позволяет восстанавливать данные из сегмента отката (корзины)
PURGE	Удаляет таблицы без возможности восстановления
RENAME	Изменяет имя существующей таблицы, представления, последовательности или частного синонима

Словарь данных *Oracle*. Словарь данных – это множество системных таблиц и объектов БД, которые хранят информацию о структуре БД и объектах внутри нее. Обновление в этих таблицах происходит в фоновом режиме, когда выполняются операторы *DDL*. В дальнейшем *Oracle* запрашивает информацию из таблиц словаря данных для синтаксического разбора любого оператора *SQL*. Информация словаря кешируется в области разделяемого пула в системной глобальной области памяти (*SGA – System Global Area*) [5]. Таблицы словаря данных содержат нормализованную информацию, которая является довольно трудной для восприятия, поэтому в *Oracle* предусмотрен набор представлений, выдающих информацию главных системных таблиц в более понятном пользователю виде. Главное представление словаря называется DICT, а вся информация о других представлениях находится в результате выполнения запроса к нему.

2.2. Стандартные типы данных *SQL* и типы данных *Oracle*

Каждый столбец БД имеет свой тип, указываемый при создании столбца. **В стандарте *SQL* определены следующие типы**:

1. Символьные: CHARACTER (*длина*); CHAR (*длина*); CHARACTER VARYING (*длина*); CHAR VARYING (*длина*); VARCHAR (*длина*); NATIONAL CHARACTER (*длина*); NATIONAL CHAR (*длина*); NCHAR (*длина*); NATIONAL CHARACTER VARYING (*длина*); NATIONAL CHAR VARYING (*длина*); NCHAR VARYING (*длина*);
2. Двоичные: BIT (*длина*); BIT VARYING (*длина*);
3. Числовые: NUMERIC; DECIMAL; DEC; INTEGER; INT; SMALLINT; FLOAT; REAL; DOUBLE PRECISION;
4. Даты/времени: DATE; TIME; TIME WITH TIME ZONE; TIMESTAMP; TIMESTAMP WITH TIME ZONE;

5. Интервальные. Тип данных INTERVAL хранит значение, представляющее промежуток времени. Типы INTERVAL делятся на два класса интервалов: «год – месяц» (INTERVAL YEAR TO MONTH) и «день – секунда» (INTERVAL DAY TO SECOND).

Для символьных типов возможно указание фразы CHARACTER SET {*set_name* | *using_form*}, устанавливающей используемый набор символов.

Приведем краткое описание наиболее часто используемых типов данных:

CHAR (*длина*) – текстовая строка фиксированной длины (в памяти выделяется место под всю строку);

VARCHAR (*длина*) – текстовая строка переменной длины, содержащая не более *num* символов (на диске выделяется место в зависимости от длины строки);

INTEGER или INT – целое;

NUMERIC – число с плавающей точкой, возможно определение числа знаков после запятой;

DECIMAL или DEC – число с плавающей точкой, возможно задание минимального значения точности;

FLOAT – число с плавающей точкой, позволяющее задавать точность (количество знаков после запятой);

REAL – число с плавающей точкой, точность которого определяется реализацией;

DATE – тип даты;

TIME WITH TIME ZONE – тип времени, содержащий поля, описывающие сдвиг зонального времени.

Типы данных, описывающие дату и время, состоят из нескольких полей, в которых хранятся части даты времени.

Так, тип DATE содержит поля YEAR, MONTH и DAY.

Тип TIME содержит поля HOUR, MINUTE и SECOND.

Тип TIMESTAMP содержит как поля даты, так и поля времени.

Значение типа TIMESTAMP записывается следующим образом: '10.1.2003 08:30:00'. Порядок следования полей при написании даты, как правило, определяется установками ОС компьютера и может быть изменен в реестре.

Сервер базы данных *Oracle* имеет следующие встроенные типы данных:

1. Числовые:

NUMBER (*p*, *s*) – число с точностью *p* и масштабом *s*. Точность может варьироваться от 1 до 38, масштаб – от –84 до 127.

BINARY_FLOAT – 32-битное число с плавающей точкой. Для хранения значения такого типа требуется 5 байт, включая байт для длины.

BINARY_DOUBLE – 64-битное число с плавающей точкой. Для хранения значения такого типа требуется 9 байт.

FLOAT (*p*) – тип данных, удовлетворяющий стандарту *ANSI*. Тип данных FLOAT описывает число с плавающей точкой с заданной двоичной точностью *p*. По умолчанию точность этого типа данных – 126 двоичных цифр или 38 десятичных.

INTEGER – эквивалентен типу данных NUMBER (*p*, 0).

2. Символьные:

CHAR – символьные данные постоянной длины, задаваемой в байтах или символах. Максимальный размер – 2000 байт или символов. Размер по умолчанию и минимальный размер – 1 байт.

VARCHAR2 (*длина*) – символьные строковые данные переменной длины, ограниченной размером в байтах или символах. Максимальный размер – 4000 байт. Размер должен всегда указываться при описании этого типа.

NCHAR (*длина*) – тип данных, хранящий символьную информацию в кодировке *Unicode*. При создании таблицы длина столбцов типа NCHAR указывается в символах. Национальная кодировка символов определяется при создании БД. Максимальная длина столбца зависит от национальной кодировки. Длина задается числом символов. Максимальный размер столбца – 2000 байт. Если вставляемое значение короче длины столбца, СУБД добавляет к нему справа пробелы, пока значение в столбце не будет иметь заданный размер. Нельзя вставить данные типа CHAR в столбец типа NCHAR, а также нельзя вставить данные типа NCHAR в столбец типа CHAR.

NVARCHAR2 (*длина* [BYTE | CHAR]) – тип данных NVARCHAR2 может хранить данные только в кодировке *Unicode*. Он схож с типом NCHAR, за исключением того, что максимальная длина NVARCHAR2 – 4000 байт и к значению в этом столбце не добавляются пробелы.

LONG – символьные данные переменной длины, размером до 2 Гбайт (до 2^{31}–1 байт). Тип данных LONG больше не рекомендуется использовать; вместо него применяется тип данных CLOB.

ROWID – уникальный адрес строки в таблице, представленный в 64-символьной системе кодирования; этот тип данных специально предназначен для возврата значения псевдостолбца ROWID и содержит информацию о физическом адресе строки в памяти (подробнее в подразделе 7.1). Например, можно выполнить запрос:

```
SELECT ROWID, SUBSTR(ROWID,15,4) "FILE",
SUBSTR(ROWID,1,8) "BLOCK", SUBSTR(ROWID,10,4) "ROW" FROM staff;
```

Таблица *staff* в данном примере и далее принадлежит учебной БД, которая описана в конце раздела (см. рис. 2.3 на с. 63) и должна быть создана самостоятельно.

UROWID – строка в 64-символьной системе кодирования, представляющая логический адрес строки в индексно-организованной таблице. В описании типа может быть указан размер. Максимальный размер и размер по умолчанию – 4000 байт.

3. Двоичные:

RAW (*размер*) – необработанные двоичные данные с длиной, заданной параметром размер. Максимальный размер – 2000 байт (он должен быть задан).

LONG RAW – необработанные («сырые») двоичные данные длиной до 2 Гбайт.

4. Дата/время:

DATE – дата в диапазоне от 1 января 4712 г. до нашей эры до 31 декабря 9999 г. нашей эры; в дате хранится время с точностью до секунды. Формат данных определяется настройками сервера. Например, увидеть текущие дату и время в *SQL*Plus* можно, выполнив команду:

```
SELECT TO_CHAR (SYSDATE, 'Dy DD-Mon-YYYY HH24:MI:SS') AS
"Current Time" FROM dual;
```

Шаблон даты, который установлен по умолчанию в *SQL*Plus* или любом другом клиентском приложении сервера *Oracle*, можно всегда увидеть, выполнив команду:

```
SELECT SYSDATE FROM dual;
```

TIMESTAMP [(*fractional_seconds_precision*)] [WITH TIME ZONE] | [WITH LOCAL TIME ZONE] – хранит значение года, месяца, дня, а также время в часах, минутах и секундах с точностью в долях секунды. Значение параметра *fractional_seconds_precision* определяет число цифр в дробной части поля секунд (доли секунды), может быть задано в диапазоне от 0 до 9. Значение по умолчанию 6. WITH TIME ZONE хранит вместе со значением даты и времени информацию о часовом поясе с точностью до девяти десятичных позиций, WITH LOCAL TIME ZONE хранит значение даты и времени с точностью до девяти десятичных позиций. Значения этого типа автоматически преобразуются между часовым поясом БД и местным (сеансовым) часовым поясом. При хранении в БД значения преобразуются к часовому поясу БД, а при выборке они преобразуются к местному (сеансовому) часовому поясу.

5. Интервальные:

INTERVAL – используются для хранения разницы между двумя значениями типа дата/время. Используются следующие интервалы:

INTERVAL YEAR (*year_precision*) TO MONTH – данные этого типа хранят период времени, выраженный в годах и месяцах, где *year_precision* – количество разрядов в поле YEAR типа дата/время. Допустимые значения находятся в диапазоне от 0 до 9. Значение по умолчанию равно 2.

INTERVAL DAY (*day_precision*) TO SECOND (*fractional_seconds_precision*) – в этом типе данных хранится период времени, выраженный в днях, часах, минутах и секундах, где *day_precision* – максимальное количество разрядов в поле DAY типа дата/время (допустимые значения находятся в диапазоне от 0 до 9, значение по умолчанию равно 2), *fractional_seconds_precision* – количество разрядов в дробной части поля SECOND, допустимые значения находятся в диапазоне от 0 до 9, значение по умолчанию равно 6.

6. Большие объекты:

BLOB – двоичный большой объект.

CLOB – символьный большой объект, содержащий однобайтные и многобайтные символы. Поддерживаются кодировки символов постоянной и переменной длины. В обоих случаях используется кодировка БД, применяемая для хранения данных типа CHAR.

NCLOB – символьный большой объект; символы в кодировке *Unicode*. Поддерживаются кодировки символов постоянной и переменной длины. В обоих случая используется национальная кодировка БД, применяемая для хранения данных типа NCHAR.

Максимальный размер всех типов данных LOB (BLOB, CLOB и NCLOB) составляет (4 Гбайта – 1) · (величина параметра CHUNK), где CHUNK – необязательный атрибут, который можно установить в описании LOB. CHUNK задает количество байт, выделяемое для манипуляций с данными LOB. Если величина CHUNK не кратна размеру блока данных БД, тогда производится округление до следующего кратного. Например, если размер блока БД равен 2048, а величина CHUNK – 2050, тогда БД выделит 4096 байт (2 блока). Максимальное значение – 32 768 (32К), представляющее собой максимально допустимый размер блока БД *Oracle*. По умолчанию величина CHUNK совпадает с размером одного блока БД *Oracle*.

BFILE – содержит указатель (*locator*) на двоичный файл, хранимый вне БД. Предоставляется возможность потокового побайтного ввода-вывода для доступа к внешним LOB-объектам, хранимым на сервере БД. Максимальный размер – 4 Гбайта.

Соответствие между типами данных *Oracle* и типами данных *ANSI*/*ISO* представлено в табл. 2.2.

Таблица 2.2

Соответствие между типами данных *Oracle* и типами данных *ANSI/ISO*

Тип данных *ANSI/ISO*	Тип данных *Oracle*
CHARACTER (*n*), CHAR (*n*)	CHAR (*n*) – символьное поле длиной до 2000 символов; по умолчанию размер 1 байт
CHARACTER VARYING (*n*), CHAR VARYING (*n*)	VARCHAR2 (*n*) – символьное поле переменной длины, до 4000 символов
NATIONAL CHARACTER (*n*), NATIONAL CHAR (*n*), NCHAR (*n*)	NCHAR (*n*)
NATIONAL CHARACTER VARYING(*n*) NATIONAL CHAR VARYING (*n*) NCHAR VARYING(*n*)	NVARCHAR2 (*n*)
NUMERIC (*p*, *s*), DECIMAL (*p*, *s*)	NUMBER (*p*, *s*) – числовое поле длиной до 38 позиций
INTEGER, INT, SMALLINT	NUMBER (38)
FLOAT, DOUBLE PRECISION, REAL	NUMBER

Логический тип данных BOOLEAN в таблицах БД *Oracle* не применяется.

2.3. Создание таблиц (инструкция CREATE TABLE)

Таблица является базовой структурой реляционной модели. Инструкция CREATE TABLE определяет структуру новой таблицы БД и подготавливает ее к приему данных. Предложения инструкции задают основные элементы определения таблицы, такие, как названия и тип данных, различные виды ограничений, место физического хранения таблицы на диске, а также распределяют его при работе с таблицей.

CREATE TABLE [схема.]*имя_таблицы* (
{*имя_столбца тип_данных_столбца* [DEFAULT *выражение*]
[*ограничение целостности столбца*]},…
[*ограничение_целостности_таблицы*], …)
[{CLUSTER *имя_кластера* (*имя_столбца*[,…])|
{PCTFREE *целое* | PCTUSED *целое* | INITRANS *целое* | MAXTRANS *целое* | TABLESPACE *имя табличной области* | STORAGE *размер памяти* |
{RECOVERABLE | UNRECOVERABLE}}…]
[PARALLEL *возможность_параллельной_обработки*]
[{ENABLE *проверяемые_ограничения_целостности* |
DISABLE *игнорируемые_ограничения_целостности*}]
[AS *запрос*]
[CACHE | NOCACHE]

где DEFAULT указывает на то, что при вводе данных соответствующему столбцу будет присвоено значение, определенное выражением, если в операторе INSERT не указано явно другое значение столбца. Тип данных выражения должен соответствовать типу данных столбца, и выражение не должно содержать ссылок на другие выражения;

CLUSTER указывает привязку столбцов таблицы к кластеру. Обычно столбцы кластера создают из элементов первичного ключа. В СУБД *Oracle* кластер – это специализированный объект БД, используемый для физически совместного хранения одной или нескольких таблиц, которые часто соединяются вместе в *SQL*-запросах. Кластеры хранят взаимосвязанные строки разных таблиц вместе в одних и тех же блоках данных, что позволяет сократить количество операций дискового ввода-вывода и улучшить время доступа для соединений таблиц, входящих в кластер. Только после создания кластера в нем можно создавать таблицы, а перед тем как добавлять строки в кластеризованные таблицы, необходимо создать индекс кластера;

PCTFREE определяет процент пространства блока, который резервируется для модификации таблицы. Допустимые значения от 0 до 99. Значение по умолчанию 10, т. е. при заполнении каждого блока 10 % пространства остается неиспользованным данными, оно используется для записи данных о модификации строк таблицы;

PCTUSED определяет минимальный процент использования пространства блока, при котором в него вводятся данные, по умолчанию 40, т. е. если в блоке занято 40 % пространства, в него вводятся данные при выполнении вставки. Сумма значений параметров PCTFREE и PCTUSED не должна превышать 100;

INITRANS определяет начальное число параллельных транзакций, которые могут выполняться для модификации данных блока. По умолчанию – 1;

MAXTRANS определяет максимальное число параллельных транзакций, которые могут выполняться для модификации данных блока. В большинстве случаев явное задание этих параметров не требуется;

TABLESPACE определяет имя табличного пространства, в котором будет размещена таблица. Если значение параметра не определено, то таблица размещается в табличном пространстве, заданном по умолчанию для пользователя, который является владельцем схемы;

STORAGE определяет объем внешней памяти, выделяемый под таблицу. Для больших таблиц целесообразно явно выделять требуемую память, чтобы уменьшить число запросов на динамическое выделение пространства;

RECOVERABLE и UNRECOVERABLE используются для управления записью в журнальный файл контрольной информации. Значение UNRECOVERABLE может быть использовано только с ключевым словом AS *запрос*, при этом операция создания таблицы выполняется быстрее за счет исключения записи управляющей информации в журнал, но при этом восстановление операции создания таблицы при сбое становится невозможным;

ENABLE указывает на включение ограничений целостности для данной таблицы. Соответствующее ограничение целостности должно быть определено в данном предложении создания таблицы. По умолчанию все ограничения включены;

DISABLE указывает на выключение ограничений целостности для данной таблицы;

AS *запрос* включает в создаваемую таблицу строки, являющиеся результатом выполнения запроса. Столбцы, извлекаемые запросом, должны соответствовать столбцам, перечисленным при создании. Тип данных и длина столбцов наследуются из запроса, также можно опустить и имена столбцов, если они в запросе имеют полностью определенные уникальные имена. В запросе нельзя употреблять предложения ORDER BY и FOR UPDATE OF. Необходима определенная осторожность при использовании вставки строк через подзапрос и определении ограничений целостности в том же предложении, так как если результат запроса не соответствует ограничениям, таблица не создается и возвращается сообщение об ошибке;

CACHE указывает на то, что блоки, выбираемые из таблицы, помечаются в системном кеше, как наиболее используемые. Рекомендуется для маленьких таблиц, используемых для преобразования кодов в значения. По умолчанию используется NOCACHE, для которого выбранные блоки помещаются в конец таблицы частот обращений к кешу;

ограничение_целостности_столбца подразумевает обязательное наличие данных (NOT NULL) в поле таблицы. Это ограничение требует обязательного присутствия в столбце некоторого значения, т. е. не допускаются значения NULL при заполнении таблицы. Обязательное наличие данных в столбце определяется разработчиком на стадии проектирования и зависит только от его смысловой нагрузки в данной таблице.

После выполнения инструкции CREATE TABLE в БД появляется новая таблица, которой присваивается имя, указанное в инструкции, и владельцем которой становится создавший ее пользователь. Имя таблицы должно быть идентификатором, допустимым в *SQL*, и не должно конфликтовать с именами других таблиц в схеме. Созданная таблица является пустой, данные записываются в нее с помощью *DML* инструкции INSERT INTO.

Задание ограничений целостности на таблицу осуществляется на основе следующей синтаксической конструкции:

[CONSTRAINT *имя_ограничения*] *тип_ограничения*

где *тип ограничения* может быть одним из следующих:

1. Первичный ключ (PRIMARY KEY) – это ограничение требует, чтобы вводимые в столбец значения были уникальными и отличными от пустых (NULL), поскольку они будут использоваться в качестве первичного ключа, однозначно идентифицирующего запись; первичный ключ определяется для таблицы только единожды и поддерживает ее целостность, также этот столбец по умолчанию является индексированным. Ключ может быть простым (задается одним полем) или составным (задается несколько полей через запятую).

Синтаксис объявления первичного ключа на уровне таблицы:

PRIMARY KEY ({*имя столбца*, …})

2. Уникальность (UNIQUE) – это ограничение требует, чтобы вводимые в столбец значения в рамках одной таблицы были уникальными, поэтому также создается с уникальным индексом, условие может быть простым и составным. Однако в столбце с таким ограничением может находиться несколько значений NULL. Синтаксис объявления:

UNIQUE ({*имя_столбца*, …})

3. Условие на значение CHECK – это ограничение позволяет подвергнуть определенной проверке вставляемое в столбец значение; если условия, наложенные на вставляемые значения, не выполняются, то значения в столбец не помещаются. Условие отбора соответствует всем правилам написания условий, возможных в предложении WHERE инструкции SELECT. Синтаксис объявления:

CHECK ({*условие отбора*})

4. Внешний ключ таблицы (FOREIGN KEY) позволяет установить взаимосвязь значений указанного столбца со значениями столбца другой таблицы при помощи ключевого слова REFERENCES. Таблица, на чей столбец ссылается другая таблица, называется главной или родительской, а таблица, ссылающаяся на нее, – подчиненной или дочерней. Взаимосвязь обеспечивается использованием следующей синтаксической конструкции:

FOREIGN KEY *имя_столбца* REFERENCES *имя_таблицы-родителя* [(*имя_столбца*)] [ON DELETE CASCADE | ON DELETE SET NULL | ON DELETE NO ACTION]

При внесении значения в столбец создаваемой таблицы система будет автоматически проверять наличие аналогичного значения в указанном столбце таблицы-родителя. При этом естественно, что для обеспечения однозначности устанавливаемой взаимосвязи все значения, находящиеся в столбце, на которые производится ссылка, должны иметь ограничение UNIQUE или PRIMARY KEY. Имя столбца в таблице-родителе может не указываться, если оно совпадает с именем в таблице-потомке.

В *Oracle* используется только три правила удаления из четырех стандартных, которые будут рассмотрены далее:

– ON DELETE CASCADE удаляет запись и вместе с ней все дочерние;

– ON DELETE SET NULL устанавливает пустые значения для дочерних строк в случае удаления родительской;

– ON DELETE NO ACTION (установлено по умолчанию) запрещает выполнение операций удаления или обновления родительской записи, если у нее есть дочерние.

Предписанные стандартом *SQL* аналогичные правила для UPDATE в СУБД *Oracle* не поддерживаются.

Определения столбцов. Определения столбцов представляют собой заключенный в скобки список, элементы которого отделены друг от друга запятыми. Порядок следования определений столбцов в списке соответствует расположению столбцов в таблице. При описании столбца указывается следующая информация:

– *имя столбца*, которое используется для ссылки на столбец в инструкциях *SQL*. Каждый столбец в таблице должен иметь уникальное имя, но в разных таблицах имена столбцов могут совпадать;

– *тип данных столбца*, указывающий, данные какого вида должны храниться в столбце. Если для типа данных требуется дополнительная информация (например, размерность или число цифр после запятой), то она указывается в скобках за ключевым словом, задающим тип данных;

– *значение по умолчанию*, которое заносится в столбец после ключевого слова DEFAULT в том случае, если в инструкции INSERT для таблицы не будет указано другое значение данного столбца;

– *указание на то, обязательно ли столбец должен содержать данные*. Ограничение NOT NULL, если оно задано, следит за обязательным наличием данных в столбце при заполнении таблицы, в противном случае значения NULL допускаются.

В *Oracle*, начиная с версии 12*c*, появились столбцы с автоинкрементными значениями (*Identity Columns*). Механизм работы таких столбцов основан на автоматическом создании последовательности с системным именем (она хранится в словаре данных USER_SEQUENCES) и фоновом выполнении триггера при вставке значений в таблицу (триггер недоступен), что приводит к значительному ускорению работы по сравнению с триггерами, которые явно создавались и хранились в БД при старом подходе.

Чтобы создать колонку с автоинкрементными значениями, в инструкции CREATE TABLE при объявлении столбца указывается следующий синтаксис:

GENERATED [ALWAYS | BY DEFAULT [ON NULL]]
AS IDENTITY [(*свойства_идентификации*)]

С помощью опций можно указывать, когда вставлять значения и задавать их свойства:

– GENERATED ALWAYS AS IDENTITY – столбец вообще нельзя указывать в списке полей INSERT;

– GENERATED BY DEFAULT AS IDENTITY – столбец можно использовать в списке INSERT, при этом вставится указанное в VALUES значение, если оно не NULL (в этом случае будет сообщение об ошибке);

– GENERATED BY DEFAULT ON NULL AS IDENTITY – столбец можно использовать в списке INSERT, вставляется либо указанное значение, либо, если указан NULL, будет выбран следующий номер из последовательности;

– *свойства_идентификации* – это перечень ключевых слов, используемых при создании последовательности для задания ей необходимых свойств.

Автоинкрементные столбцы имеют ограничение NOT NULL. Также необходимо понимать, что если не указывать для такого столбца ограничение PRIMARY KEY, то при использовании многострочной инструкции INSERT в таблицу будут вставлены записи с одним и тем же значением.

Например, создадим таблицу с первичным ключом, использующим инкрементный столбец, который не нужно указывать при вставке:

```
CREATE TABLE tab (
c1 NUMBER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
c2 VARCHAR2 (10));
```

Или создадим таблицу *tab*1, где значения столбца при вставке могут задаваться самостоятельно, а последовательность будет работать только при указании NULL, при этом настроим последовательность на начальное значение 100, шаг 50 и отменим кеширование значений.

```
CREATE TABLE tab1 (
c1 NUMBER GENERATED BY DEFAULT ON NULL AS IDENTITY (START WITH 100 INCREMENT BY 50 NOCACHE), c2 VARCHAR2 (10));
```

Для нахождения имени автоматически создаваемой последовательности в словаре БД можно использовать команду:

```
SELECT TABLE_NAME, COLUMN_NAME, DATA_DEFAULT
FROM USER_TAB_COLUMNS WHERE TABLE_NAME = 'TAB1';
```

где USER_TAB_COLUMNS – это одно из представлений словаря данных.

В столбце DATA_DEFAULT хранится информация в виде *имя_последовательности*.NEXTVAL. Такой подход к работе с автоинкрементными столбцами позволил использовать значения любой пользовательской последовательности в качестве значений по умолчанию при создании таблиц, что ранее в версии 12*c* было невозможно. Увидеть следующее значение последовательности можно выполнив команду:

```
SELECT ISEQ$$_89900173.NEXTVAL FROM DUAL;
```

где ISEQ$$_89900173 – это системное имя последовательности из словаря данных.

Определение первичного и внешнего ключей. Кроме определений столбцов таблицы, в инструкции CREATE TABLE указывается информация о первичном ключе таблицы и ее связях с другими таблицами БД. Эта информация содержится в предложениях PRIMARY KEY и FOREIGN KEY.

В предложении PRIMARY KEY задается столбец или столбцы, которые образуют первичный ключ таблицы. Как известно, этот столбец (или комбинация столбцов) служит в качестве уникального идентификатора строк таблицы. СУБД автоматически следит за тем, чтобы первичный ключ каждой строки таблицы имел уникальное значение. Кроме того, в столбцах, использующихся в качестве первичного ключа, не должны допускаться значения NULL (имеют ограничение NOT NULL по умолчанию).

В предложении FOREIGN KEY задается внешний ключ таблицы и определяется столбец связи с другой таблицей. В нем может быть указано:

– необязательное имя связи для этого отношения после ключевого слова CONSTRAINT; оно практически не используется в инструкциях *SQL*, но может появляться в сообщениях об ошибках и потребуется, если будет необходимо удалить внешний ключ и разорвать связь между таблицами;

– столбец или столбцы создаваемой таблицы, которые уже определены и будут использоваться как внешний ключ для связи с другой таблицей-предком (определяемая таблица в данном отношении является потомком);

– необязательное правило удаления для данного отношения, которое определяет действие, предпринимаемое при удалении строки-предка.

– имя столбца в таблице-предке всегда можно указать в скобках после имени таблицы-предка, но обязательным оно является только если связываемые столбцы в таблице-предке и таблице-потомке имеют различные имена.

Когда СУБД выполняет инструкцию CREATE TABLE, она сравнивает определение каждого внешнего ключа с определениями связанных таблиц. СУБД проверяет, соответствуют ли друг другу внешний ключ и первичный ключ в связанных таблицах как по числу столбцов, так и по типу данных, включая и размер поля. Для того чтобы такая проверка была возможна, связанная таблица на момент создания внешнего ключа уже должна быть определена.

Если две или более таблиц образуют **ссылочный цикл**, то для первой создаваемой таблицы невозможно определить внешний ключ, так как связанная с нею таблица еще не существует. СУБД откажется выполнять инструкцию CREATE TABLE, выдав сообщение о том, что в определении таблицы присутствует ссылка на несуществующую таблицу. В этом случае необходимо создать таблицу без определения внешнего ключа и добавить данное определение позднее с помощью инструкции ALTER TABLE. Такая же ситуация может возникнуть и в том случае, когда таблица ссылается сама на себя.

Условия уникальности и ограничения на значения столбцов. В соответствии со стандартом условие уникальности значений в каком-либо столбце, не выбранном в качестве первичного ключа, задается в предложении UNIQUE инструкции CREATE TABLE. Это дополнительное ограничение, которое позволяет СУБД следить за уникальностью данных в указанном столбце при заполнении таблицы при помощи автоматически создаваемого уникального индекса. Однако, в отличие от первичного ключа, поле может оставаться пустым, если явно не задано ограничение NOT NULL.

Ограничение CHECK задает дополнительные ограничения в инструкции CREATE TABLE. Оно содержит условие на значения (идентично условию отбора WHERE в запросе на выборку), проверяемое всякий раз при попытке модификации содержимого таблицы (с помощью инструкций INSERT, UPDATE или DELETE). Если после модификации условие остается истинным, такое изменение допускается; в противном случае СУБД отвергает изменения и выдает сообщение об ошибке. Необходимо отметить, что в *Oracle* можно использовать в данном ограничении только детерминированные функции, в связи с чем сравнение значений поля с системной недетерминированной переменной SYSDATE невозможно, но есть возможность применить в обучающих целях следующий подход:

```
CREATE TABLE t (x DATE,
y date DEFAULT SYSDATE,
CONSTRAINT x_y CHECK (x >= y));
```

На практике, конечно, такой подход не применяется, а задача решается путем написания элементарного *DML*-триггера.

Создать таблицу с ограничениями уникальности и проверки значений можно, например, следующим образом:

```
CREATE TABLE salesreps (sale_n INTEGER,
name VARCHAR2 (15),
position VARCHAR2 (10),
sex CHAR (1),
salary NUMBER (6,2),
age DATE,
CONSTRAINT salesreps_name_unique UNIQUE (name),
CONSTRAINT salesreps_pk PRIMARY KEY (sale_n),
CHECK (sex IN ('m', 'f')),
CHECK (salary<=500.00));
```

Если необходимо создать таблицу *orders* в соответствии со схемой данных, представленной на рис. 2.1 (предполагается, что другие таблицы схемы уже есть в БД), то можно сделать это следующим образом:

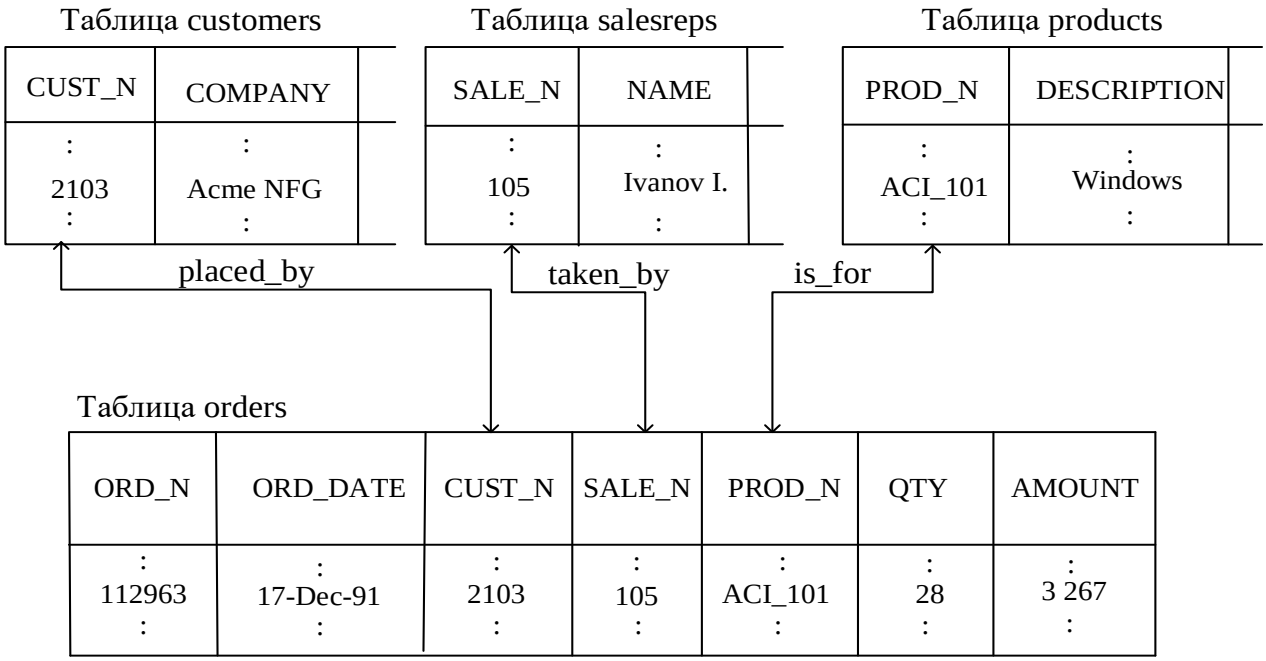


Рис. 2.1. Таблица *orders* со связями

```
CREATE TABLE orders (order_n INTEGER,
ord_date DATE NOT NULL,
cust_n INTEGER,
sale_n INTEGER,
prod_n VARCHAR2(10),
qty INTEGER,
amount INTEGER,
PRIMARY KEY (order_n),
CONSTRAINT placed_by FOREIGN KEY (cust_n) REFERENCES customers,
CONSTRAINT taken_by FOREIGN KEY (sale_n) REFERENCES salesreps,
CONSTRAINT is_for FOREIGN KEY (prod_n) REFERENCES products);
```

Если первичный или внешний ключ включает в себя только один столбец либо если условие уникальности или условие на значение касаются одного столбца, стандарт разрешает использовать сокращенную форму записи инструкции CREATE TABLE с написанием ограничений на уровне столбцов. Например, создание некой таблицы *offices* при существовании *regions* выглядит так:

```
CREATE TABLE offices (office_n INTEGER PRIMARY KEY,
city VARCHAR2(15) NOT NULL UNIQUE,
region_n VARCHAR2(10) NOT NULL REFERENCES regions);
```

2.4. Удаление таблицы (инструкции DROP TABLE и PURGE)

С течением времени структура БД изменяется. Ненужные таблицы можно удалять посредством инструкции DROP TABLE. Стандартная синтаксическая диаграмма следующая:

DROP TABLE [*схема.*]*имя_таблицы* [CASCADE | RESTRICT];

Инструкция содержит имя удаляемой таблицы. Обычно пользователь удаляет одну из своих собственных таблиц и указывает в инструкции простое имя таблицы. Имея соответствующее разрешение, можно также удалить таблицу другого пользователя, но в этом случае необходимо указать полное имя таблицы.

В соответствии со стандартом инструкция DROP TABLE может включать в себя либо параметр CASCADE, либо RESTRICT, которые определяют, как влияет удаление таблицы на другие объекты БД (например, представления), зависящие от этой таблицы. Если задан параметр RESTRICT и в БД имеются объекты, которые содержат ссылку на удаляемую таблицу, то выполнение инструкции закончится неуспешно. В большинстве коммерческих СУБД допускается применение инструкции DROP TABLE без каких-либо параметров.

Синтаксическая диаграмма DROP TABLE в *Oracle* имеет следующий вид:

DROP TABLE [*schema*.]*имя_таблицы*[CASCADE CONSTRAINTS] [PURGE];

Обычно с таблицей в БД связано несколько объектов, например, индекс, создаваемый первичным ключом, или ограничение UNIQUE, налагаемое на столбцы таблицы. При удалении таблицы *Oracle* автоматически удаляет и любой связанный с ней индекс.

Конструкция CASCADE CONSTRAINTS позволит удалить таблицу, если на ее столбцы определены ссылки внешними ключами других таблиц (внешний ключ, ссылающийся на значения столбцов «собственной» таблицы, не препятствует ее удалению). При этом СУБД удалит сначала «мешающее» правило внешнего ключа, а столбцы другой, оставшейся таблицы, в результате сохранят свои значения, но они уже не будут обременены ограничением ссылочной целостности. Фактически использование CASCADE CONSTRAINTS равносильно последовательному удалению всех правил внешнего ключа, ссылающихся на

удаляемую таблицу (таковых может быть несколько), и выполнению простой команды DROP TABLE.

Конструкция PURGE появилась в версии *Oracle* 10*g* и изменила смысл команды DROP. Теперь после ее выполнения описание и данные таблицы продолжают храниться на своих местах, но под новыми, присвоенными системой автоматически именами. Для пользователя таблица удалена, однако все, что нужно для ее восстановления, если 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 (словаря данных) USER_RECYCLEBIN (публичный синоним – RECYCLEBIN) командой:

```
SELECT OBJECT_NAME, ORIGINAL_NAME, DROPTIME
FROM USER_RECYCLEBIN;
```

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

FLASHBACK TABLE *имя_таблицы* TO BEFORE DROP;

При этом у восстановленной таблицы будут отсутствовать, возможно, имевшиеся до удаления ограничения целостности, а индексы, хотя и восстановятся, но уже с новыми системными именами.

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

PURGE TABLE *имя_таблицы*;

Можно очистить корзину целиком:

PURGE RECYCLEBIN;

Чтобы таблица удалялась сразу без возможности восстановления, следует использовать предложение PURGE в инструкции DROP:

DROP TABLE *имя_таблицы* PURGE;

2.5. Изменение определения таблицы (инструкция ALTER TABLE)

Изменения в определении таблицы можно осуществить с помощью инструкции ALTER TABLE. Данная инструкция, как и DROP TABLE, обычно применяется пользователем по отношению к своим собственным таблицам. Однако, имея соответствующее разрешение и используя полное имя таблицы, можно изменять таблицы и других пользователей.

Синтаксис инструкции следующий:

ALTER TABLE *имя_таблицы*
{ADD *определение_столбца* |
MODIFY *новое_определение_столбца* |
DROP COLUMN *имя_столбца* [CASCADE CONSTRAINT] |
ADD [CONSTRAINT]{*определение_первичного_ключа* |
определение внешнего ключа | *условие на значение* | *условие уникальности*}|
DROP CONSTRAINT *имя_ограничения* [CASCADE]|
MODIFY *имя_ограничения* ENABLE | DISABLE |
RENAME *имя_столбца* TO *новое_имя_столбца* };

Таким образом, инструкция ALTER TABLE может:

– добавить в таблицу определение нового столбца;
– удалить существующий столбец из таблицы;
– изменить для какого-либо столбца значение по умолчанию;
– добавить или удалить первичный ключ таблицы;
– добавить или удалить внешний ключ таблицы;
– добавить или удалить условие уникальности;
– добавить или удалить условие на значения;
– переименовать столбец;
– включить или выключить ограничение.

В различных коммерческих СУБД некоторые из стандартных возможностей ALTER не используются вообще, либо используются специфические предложения, либо добавляются совершенно новые возможности изменения таблиц. Так, вместо регламентированного стандартом предложения для изменения свойств столбца ALTER в *Oracle* использует ключевое слово MODIFY, а также добавлено новое предложение для переименования существующих столбцов.

Чтобы переименовать столбец в таблице *Viewing*, используется запись:

ALTER TABLE Viewing RENAME COLUMN comment_o TO comment1;

Также стандарт требует, чтобы инструкция ALTER TABLE применялась для единичного изменения таблицы. Например, для добавления столбца и определения нового внешнего ключа потребуются две различные инструкции. В некоторых СУБД, в том числе *Oracle*, это ограничение ослаблено и в некоторых случаях допускается присутствие нескольких предложений в одной инструкции ALTER TABLE.

Добавление и удаление столбца. Чаще всего инструкция ALTER TABLE применяется для добавления столбца в существующую таблицу. Предложение с определением столбца в инструкции ALTER TABLE имеет точно такой же вид, как и в инструкции CREATE TABLE. Новое определение добавляется в конец определений столбцов таблицы, и в последующих запросах новый столбец будет крайним справа. СУБД обычно предполагает, что новый столбец во всех существующих строках содержит значения NULL, поэтому нельзя объявлять столбец просто как NOT NULL, поскольку СУБД подставляла бы в существующие

строки значения NULL, нарушая тем самым заданное условие. Однако, если все-таки, необходимо, чтобы новый столбец не содержал пустые значения, его добавляют с указанием значения по умолчанию. В действительности, когда добавляется новый столбец, СУБД не заносит во все существующие строки нового столбца значения NULL или значения по умолчанию. СУБД обнаруживает тот факт, что строка слишком коротка для нового определения таблицы, только при выборке этой строки пользователем, и расширяет ее значениями NULL или значениями по умолчанию непосредственно перед выводом на экран или передачей в программу пользователя.

Добавить новый столбец в таблицу *staff*:

```
ALTER TABLE staff ADD email VARCHAR2(10);
```

или можно добавить этот столбец с ограничением на ввод пустых значений:

```
ALTER TABLE staff ADD email VARCHAR2(10) default '1@url' not null;
```

С помощью инструкции ALTER TABLE можно удалить из существующей таблицы один или несколько столбцов, если в них больше нет необходимости. Например, следующая инструкция удаляет столбец *position* из таблицы *staff*:

```
ALTER TABLE staff DROP COLUMN position;
```

Стандарт требует, чтобы одна инструкция ALTER TABLE использовалась для удаления только одного столбца (*Oracle* в этом вопросе поддерживает стандарт), но в ряде ведущих СУБД такое ограничение снято.

Следует учитывать, что операция удаления столбца вызывает те же проблемы целостности данных, которые были описаны выше на примере обновления таблиц. В частности, при удалении столбца, являющегося первичным ключом в каком-либо отношении, связанные с ним внешние ключи становится недействительными. Похожая проблема возникает, когда удаляется столбец, участвующий в проверке ограничения на значения другого столбца.

Описанные проблемы решаются так же, как и в случае инструкций DELETE и UPDATE – с помощью правила удаления. Можно выбрать одно из двух правил:

– NO ACTION (задан неявно, по умолчанию): если с удаляемым столбцом связан какой-либо объект в БД (внешний ключ, ограничение и т. п.), инструкция ALTER TABLE завершится выдачей сообщения об ошибке и столбец не будет удален;

– CASCADE (указывается явно): любой объект БД (внешний ключ, ограничение и т. п.), связанный с удаляемым столбцом, также будет удален.

Правило CASCADE может вызвать целую «лавину» изменений, поэтому применять его следует с осторожностью. Лучше использовать правило NO ACTION, а связанные внешние ключи или ограничения обрабатывать с помощью дополнительных инструкций с необходимой логикой управления данными.

Самым простым способом избежать многих проблем является временное отключение ограничения, например, если есть именованное ограничение на уникальность телефона сотрудника, то для отключения может использоваться команда

```
ALTER TABLE staff MODIFY CONSTRAINT uni_telno_staff DISABLE;
```

Изменение первичных и внешних ключей. Еще одним распространенным случаем применения инструкции ALTER TABLE является изменение или добавление определений первичных и внешних ключей таблицы.

Используя инструкцию ALTER TABLE, определения первичного и внешних ключей (в отличие от определений столбцов) можно как добавлять в таблицу, так и удалять из нее. Предложения, добавляющие определения первичного и внешнего ключей, являются точно такими же, как в инструкции CREATE TABLE, и выполняют те же функции.

Предложения, удаляющие первичный или внешний ключи, являются довольно простыми, однако удалить внешний ключ можно только тогда, когда создаваемая им связь имеет имя. Если имя присвоено не было, то задать эту связь в инструкции ALTER TABLE невозможно. В этом случае для удаления внешнего ключа необходимо либо удалить таблицу и воссоздать ее в новом виде, либо все-таки найти в словаре данных системное имя ограничения, заданное СУБД.

Например, определим новый внешний ключ для таблицы *staff*, предполагая, что в базе появилась новая таблица *jobs*, хранящая сведения о должностях компании.

```
ALTER TABLE staff
ADD CONSTRAINT dolznost
FOREIGN KEY (position) REFERENCES jobs (job_id);
```

2.6. Условия целостности данных

Еще раз повторим и систематизируем пройденный материал с точки зрения поддержания целостности данных в реляционной БД, когда необходимо помнить о непротиворечивости и правильности хранимой информации. Для этого любая СУБД предусматривает возможность устанавливать одно или несколько условий целостности данных. Эти условия определяют, какие значения могут быть записаны в БД в результате добавления или обновления данных. Как правило, выделяют следующие условия поддержания целостности данных.

– Обязательное наличие данных. По сути хранимой информации некоторые столбцы в БД должны содержать значения в каждой строке. Можно указать СУБД, что строки в таких столбцах не могут содержать значения NULL.

– Условие на значение. Каждый столбец в БД определен на своем домене, т. е. имеет тот набор значений, которые допускается хранить в данном столбце. Можно указать СУБД, что запись значений, не входящих в определенный диапазон, в такие столбцы недопустима.

– Целостность таблицы (сущности). Первичный ключ таблицы должен в каждой строке иметь уникальное значение, отличное от значений во всех остальных строках. Можно указать СУБД, чтобы она обеспечивала целостность таблиц за счет использования первичных ключей.

– Ссылочная целостность. Каждая строка таблицы-потомка с помощью внешнего ключа связана со строкой таблицы-предка, содержащей первичный ключ, значение которого равно значению внешнего ключа. Можно указать СУБД, чтобы она обеспечивала ссылочную целостность таблиц за счет использования связи первичных и внешних ключей.

– Бизнес-правила. Обновление информации в БД может быть ограничено деловыми правилами, которым подчиняются сделки, представляемые подобными обновлениями. Можно указать СУБД, чтобы она проверяла все необходимые бизнес-правила.

– Непротиворечивость. Многие реальные процессы вызывают в БД несколько изменений одновременно. Можно указать СУБД, чтобы все необходимые действия осуществлялись как одно целое.

Обязательное наличие данных. Это условие целостности данных требует, чтобы некоторые столбцы не содержали значений NULL. Стандарт *ANSI/ISO* и большинство коммерческих СУБД поддерживают выполнение подобного условия, позволяя пользователю при создании таблицы объявить, что некоторые столбцы не могут содержать значений NULL. Условие задается как часть инструкции CREATE TABLE в виде ограничения NOT NULL.

Если на столбец наложено ограничение NOT NULL, то для выполнения этого условия СУБД обеспечивает следующее:

– ни в одной инструкции INSERT, добавляющей в таблицу строку или строки, нельзя указывать значение NULL для этого столбца; попытка добавить строку, содержащую (явно или неявно) значение NULL для такого столбца, вызовет ошибку;

– ни в одной инструкции UPDATE, обновляющей столбец, нельзя присваивать столбцу значение NULL; попытка обновить такой столбец, присвоив ему значение NULL, вызовет ошибку.

Например, в учебной БД (см. рис. 2.3 на с. 63) для каждого объекта недвижимости должен существовать владелец, сдающий этот объект. Поэтому столбец *ono* в таблице *property_for_rent* является обязательным и необходимо указать СУБД, что запись значения NULL в такой столбец недопустима.

Условия на значения. При создании таблицы за каждым столбцом закрепляется определенный тип данных или домен, и СУБД следит за тем, чтобы в столбец вводились данные только этого типа и принадлежащие домену. Понятие домена специфично для БД, хотя и имеет некоторые аналогии с подтипами в некоторых языках программирования. Наиболее правильной трактовкой понятия «домен» является его понимание как допустимого потенциального множества значений выбранного типа данных, которые считаются сравнимыми только в том случае, когда относятся к одному домену.

Ограничение на значения столбца аналогично условию отбора в предложении WHERE и возвращает значение TRUE или FALSE. Если для столбца задано ограничение, то при каждом добавлении новой строки или обновлении старой СУБД автоматически проверяет, выполняется ли ограничение для значения в этом столбце. Если оно не выполняется, то инструкция INSERT или UPDATE завершается ошибкой. Так, например, в столбце *type* таблицы *property_for_rent* могут присутствовать только два заданных значения: квартира («f») или дом «h».

Целостность таблиц (сущностей). Каждая строка таблицы должна иметь уникальное значение первичного ключа, иначе база данных потеряет свою целостность. В ранних коммерческих СУБД первичные ключи отсутствовали, но сейчас они стали повсеместно распространенными. Первичные ключи создаются с помощью инструкции CREATE или ALTER TABLE. СУБД автоматически проверяет уникальность первичного ключа для каждой инструкции INSERT или UPDATE. Попытка добавить строку с уже существующим значением первичного ключа или обновить строку таким образом, что ее первичный ключ потеряет свою уникальность, завершится выдачей сообщения об ошибке.

Задавать первичные ключи можно с помощью автоинкрементных инструкций или выбрать столбец (несколько столбцов), который семантически идентифицирует записи таблицы, например, для таблицы сотрудников можно использовать их идентификационные номера.

Ссылочная целостность. Существует четыре типа изменений БД, которые могут нарушить ссылочную целостность отношений предок/потомок. Рассмотрим каждую из этих четырех ситуаций на примере таблиц *branch* и *staff*.

– Добавление новой строки-потомка. Когда происходит добавление новой строки в таблицу *staff*, значение ее внешнего ключа *bno* должно быть равно одному из значений первичного ключа *bno* в таблице-предке *branch*. Если значение внешнего ключа не равно ни одному из значений первичного ключа, то добавление такой строки разрушит целостность БД, поскольку появится потомок без предка («сирота»). Добавление строки в таблицу *branch* не вызовет проблем; она просто станет предком без потомков.

– Обновление внешнего ключа в строке-потомке. Это та же проблема, что и в предыдущей ситуации, но выраженная в иной форме. Если внешний ключ обновляется инструкцией UPDATE, то его новое значение должно быть равно одному из значений первичного ключа в таблице-предке. В противном случае обновленная строка окажется «сиротой».

– Удаление строки-предка. Если из таблицы-предка *branch* будет удалена строка, у которой есть хотя бы один потомок, то строки-потомки станут «сиротами». Значения внешних ключей в этих строках больше не будут равны ни одному из значений первичного ключа таблицы-предка. Удаление строки из таблицы-потомка не вызовет проблем. В этом случае предок этой строки после удаления будет иметь на одного потомка меньше или останется без потомков.

– Обновление первичного ключа в строке-предке. Если происходит изменение первичного ключа некоторой строки в таблице-предке, все существующие

потомки этой строки становятся «сиротами», поскольку их внешние ключи больше не равны ни одному первичному ключу.

Средства поддержки ссылочной целостности стандарта *ANSI/ISO* позволяют обрабатывать каждую из четырех описанных ситуаций. Первая проблема решается путем проверки значений в столбцах внешнего ключа перед выполнением инструкции INSERT. Если они не равны ни одному из значений первичного ключа, то выполнение инструкции INSERT приводит к появлению сообщения об ошибке.

Вторая проблема (обновление таблицы-потомка) решается аналогично: путем проверки нового значения внешнего ключа. Если нет ни одного равного ему значения первичного ключа, инструкция UPDATE отбрасывается с выдачей сообщения об ошибке.

Третья проблема является более сложной. Предположим, например, что закрыли отделение в Гродно и надо удалить соответствующую строку из таблицы *branch*. Как в этом случае поступить со строками-потомками таблицы *staff,* зависит от ситуации:

– не удалять из БД отделение до тех пор, пока служащие не будут переведены в другое отделение;

– автоматически удалить всех соответствующих служащих из таблицы *staff*;

– в столбце *staff.bno* установить значение NULL для всех соответствующих служащих, показывая тем самым, что идентификатор их офиса неизвестен;

– в столбце *staff.bno* для соответствующих служащих установить по умолчанию некоторое значение, например, идентификатор главного отделения в Минске, указывая тем самым, что они автоматически переводятся в это отделение.

Аналогичные сложности существуют и в четвертой ситуации. Допустим, по каким-либо причинам требуется изменить идентификатор отделения в Витебске с 1 на 2. Подобно предыдущему примеру, возникает вопрос о том, как поступить со строками-потомками в таблице *staff*, представляющими служащих витебского офиса. И снова проблему можно решить четырьмя способами:

– не изменять идентификатор отделения до тех пор, пока служащие не будут переведены в другое отделение; в таком случае в таблицу *branch* следует вначале добавить строку с новым идентификатором отделения в Витебске, затем обновить таблицу *staff* и, наконец, удалить строку со старым идентификатором витебского отделения;

– автоматически обновить идентификатор отделения этих служащих в таблице *staff* для того, чтобы их строки были по-прежнему связаны с витебской строкой в таблице *branch* через ее новый идентификатор;

– в столбце *staff.bno* установить для соответствующих служащих значение NULL, показывая тем самым, что идентификатор офиса неизвестен;

– в столбце *staff.bno* установить по умолчанию для соответствующих служащих некоторое значение, например, идентификатор главного офиса в Минске.

Правила удаления и обновления. Для каждого отношения предок/потомок в БД, создаваемого внешним ключом, стандарт *SQL* позволяет задать соответствующее правило удаления и соответствующее правило обновления. Правило удаления определяет те действия, которые СУБД выполняет, когда пользователь пытается удалить строку из таблицы-предка. В соответствии со стандартом *SQL* можно задавать одно из четырех возможных правил удаления.

– RESTRICT (NO ACTION) – запрещает удаление строки из таблицы-предка, если строка имеет потомков. Следует отметить, что данное правило установлено в большинстве СУБД по умолчанию и не требует явного задания.

– CASCADE – определяет, что при удалении строки-предка все строки-потомки также автоматически удаляются из таблицы-потомка. Данное правило опасно в употреблении, в случае если существуют многоуровневые связи между таблицами.

– SET NULL – определяет, что при удалении строки-предка внешним ключам во всех ее строках-потомках автоматически присваивается значение NULL.

– SET DEFAULT – определяет, что при удалении строки-предка внешним ключам, всем ее строкам-потомкам присваивается определенное значение, по умолчанию установленное для данного столбца.

Аналогично тому, как правила удаления определяют действия СУБД при попытке удалить строку из таблицы-предка, так правила обновления определяют действия СУБД, когда пользователь пытается обновить значение первичного ключа в таблице-предке. Регламентируются четыре правила обновления, аналогичные правилам удаления.

Для одного и того же отношения предок/потомок правила удаления и обновления могут быть различными, хотя в большинстве случаев они совпадают.

В заключение следует отметить, что правила каскадного обновления в СУБД *Oracle* недействительны, поддерживается только неявный запрет на изменение строк-предков (NO ACTION), а для удалений не поддерживается только стандартное правило SET DEFAULT.

Правило RESTRICT является «одноуровневым» – в отношении предок/потомок оно затрагивает только таблицу-предок. Правило CASCADE, напротив, «многоуровневое», его следует применять с осторожностью, поскольку некорректное его использование может вызвать широкомасштабное автоматическое удаление данных. Правила каскадного обновления могут привести к подобным многоуровневым обновлениям, если внешний ключ в таблице-потомке одновременно является и ее первичным ключом. На практике такая ситуация встречается не часто, поэтому каскадное обновление обычно не имеет таких далеко идущих последствий, как каскадное удаление.

Бизнес-логика. Все перечисленные выше ограничения используются СУБД для сохранения непротиворечивости и правильности хранимой информации уже на стадии создания таблицы, кроме этого условия целостности поддерживает так называемая бизнес-логика или деловые правила. Например, организация может установить деловое правило, запрещающее сотруднику офиса работать одновременно более чем с пятью объектами.

Реализация данных правил в СУБД может осуществляться посредством автоматически выполняемых программ-триггеров, запуск которых осуществляется после выполнения операций по изменению содержимого базы. Можно, например, с помощью триггера указать СУБД, что следует проверять каждую новую строку, добавляемую в таблицу *property_for_rent*, и убеждаться, что количество объектов, закрепленных за одним сотрудником одновременно, не превышает установленное деловое правило.

Непротиворечивость данных. Многие реальные действия пользователей вызывают несколько изменений одновременно. Например, операция увольнения сотрудника должна сопровождаться переводом всех выполняемых им работ на какого-то другого сотрудника либо путем временного задания значений NULL или значений по умолчанию для столбцов других таблиц, связанных с увольняемым сотрудником. Последовательное выполнение инструкций, в совокупности решающих общую задачу по изменению информации в БД с соблюдением ее конечного непротиворечивого состояния, решается СУБД посредством поддержки механизма транзакций. Транзакция считается успешно завершенной, если каждая из операций, входящих в нее, выполнена успешно. Подтверждением успешного завершения является выполнение инструкции COMMIT. В случае неуспешного завершения хотя бы одной из операций транзакция должна отменить все произведенные ранее изменения в базе и перевести ее в начальное непротиворечивое состояние посредством инструкции ROLLBACK.

Ссылочные циклы. В связи с реализацией отношений предок/потомок в реляционных БД может возникать ситуация, когда такие отношения образуют так называемый ссылочный цикл. На рис. 2.2 представлены две таблицы, связанные между собой отношениями предок/потомок так, что они образуют ссылочный цикл.

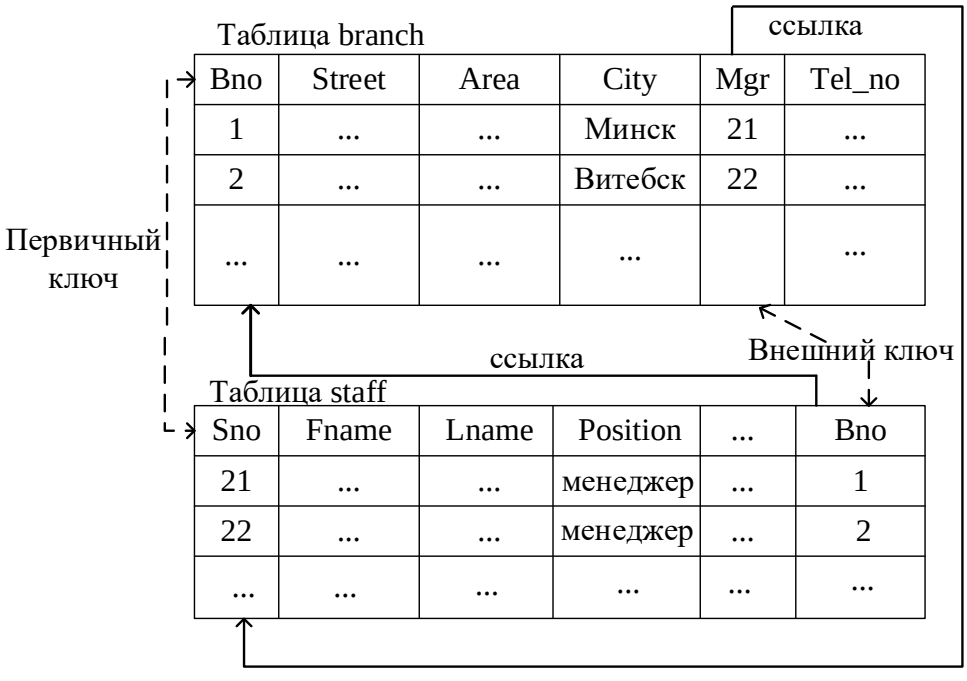


Рис. 2.2. Ссылочный цикл

Таблица *staff* содержит столбец *bno* – внешний ключ для таблицы *branch*. Таблица *branch*, в свою очередь, может содержать столбец *mgr* (с информацией об управляющих отделений) – внешний ключ для таблицы *staff*. Так эти два отношения и образуют *ссылочный цикл*. Любая строка таблицы *staff* имеет ссылку на строку таблицы *branch*, которая имеет ссылку на строку таблицы *staff*, и т. д.

Независимо от количества таблиц, принимающих в них участие, ссылочные циклы представляют особую проблему ссылочной целостности. Предположим, например, что в двух таблицах, изображенных на рис. 2.2, для первичных и внешних ключей не допускаются значения NULL. Рассмотрим, например, следующий запрос на добавление и соответствующие ему инструкции INSERT.

Поместить в БД информацию о новом сотруднике с назначением его управляющим в только что открывшемся отделении в Москве:

```
INSERT INTO staff (sno, fname, lname, salary, bno)
VALUES (23, 'Василий', 'Иванов', 500, 3);

INSERT INTO branch (bno, street, area, city, mgr, tel_no)
VALUES (3, '…', '…', 'Москва', 23, '…');
```

Выполнение первой инструкции будет безрезультатным, потому что в новой строке есть ссылка на идентификатор отделения, которого в БД еще нет. Очевидно, что изменение порядка инструкции INSERT ни к чему не приведет, так как при выполнении инструкции для таблицы *branch* встречается ссылка на идентификатор сотрудника (управляющего офисом), который еще отсутствует в таблице *staff*.

Для предотвращения подобной ситуации по крайней мере один из внешних ключей ссылочного цикла должен допускать значения NULL. Тогда ввод двух строк можно выполнить с помощью двух инструкций INSERT и одной инструкции UPDATE:

```
INSERT INTO staff (sno, fname, lname, salary, bno)
VALUES (23, 'Василий', 'Иванов', 500, NULL);

INSERT INTO branch (bno, street, area, city, mgr, tel_no)
VALUES (3, '…', '…', 'Москва', 23, '…');

UPDATE staff
SET bno=3
WHERE sno=23;
```

Как видно из данного примера, в некоторых ситуациях было бы удобно, чтобы условия ссылочной целостности вообще не проверялись до тех пор, пока не будет выполнен ряд взаимосвязанных обновлений. К сожалению, в большинстве современных СУБД отсутствует такой тип комплексной отложенной проверки. Для *Oracle* самым простым способом решения этой проблемы является программное включение/отключение всех ограничений на время добавления данных в таблицы.

Если ограничения препятствуют вводу данных и их нужно временно отменить, можно воспользоваться оператором

ALTER TABLE *имя_таблицы* DISABLE CONSTRAINT *имя_ограничения* [CASCADE];

а затем можно снова включить его оператором

ALTER TABLE *имя_таблицы* ENABLE CONSTRAINT *имя_ограничения*;

2.7. Создание представлений

Представлением называется именованный запрос на выборку, сохраненный в БД. Иногда представления называют «виртуальными таблицами», содержимое которых определяется запросом. Для пользователя базы данных представление выглядит подобно обычной таблице, состоящей из строк и столбцов. Однако в отличие от таблицы представление как совокупность значений в БД реально не существует. Строки и столбцы данных, которые пользователь видит с помощью представления, являются результатами запроса, лежащего в его основе. При создании представление получает имя, и его определение сохраняется в БД.

Представления используются по нескольким причинам:

– позволяют разным пользователям БД видеть структуру таблиц по-разному, в более удобном для них виде;

– ограничивают доступ к данным, разрешая пользователям видеть только некоторые из строк и столбцов таблицы;

– упрощают доступ к БД, показывая каждому пользователю структуру хранимых данных в наиболее подходящем для него виде.

Создание представлений. Инструкция CREATE VIEW, синтаксис которой приведен ниже, используется для создания представлений. В ней указываются имя представления и запрос, лежащий в его основе. Для успешного создания представления необходимо иметь разрешение на доступ ко всем таблицам, входящим в запрос.

CREATE OR REPLACE VIEW *имя_представления*
[(*столбец_представления* [, *столбец_представления*...])]
AS SELECT [WITH CHECK OPTION];

При необходимости в инструкции CREATE VIEW можно задать имя для каждого столбца создаваемого представления. Если указывается список имен столбцов, то он должен содержать столько элементов, сколько столбцов содержится в запросе, причем в списке задаются только имена столбцов; тип данных, длина и другие характеристики берутся из определения столбца в исходной таблице. Если список имен столбцов в инструкции CREATE VIEW отсутствует, каждый столбец представления получает имя соответствующего столбца запроса. Если в запрос входят вычисляемые столбцы или два столбца с одинаковыми именами, то без такого списка обойтись невозможно.

Когда СУБД встречает в инструкции *SQL* ссылку на представление, она отыскивает его определение, сохраненное в БД, преобразует пользовательский запрос, ссылающийся на представление, в эквивалентный запрос к исходным таблицам представления, и выполняет этот запрос. Таким образом, СУБД создает иллюзию существования представления в виде отдельной таблицы и в то же время сохраняет целостность исходных таблиц.

Если определение представления простое, то СУБД формирует каждую строку представления «на лету», извлекая данные из исходных таблиц. Если же определение сложное, СУБД приходится материализовывать представление. Это означает, что СУБД выполняет запрос, определяющий представление, и сохраняет его результаты во временной таблице. Из нее СУБД берет данные для формирования результатов пользовательского запроса, а когда временная таблица становится ненужной, удаляет ее. Но независимо от того, как СУБД выполняет инструкцию, являющуюся определением представления, для пользователя результат будет одним и тем же. Ссылаться на представление в инструкции *SQL* можно так же, как если бы оно было реальной таблицей БД. После определения представления к нему можно обращаться с помощью инструкции SELECT, как к обычной таблице:

SELECT * FROM *имя_представления*;

Также можно изменять, удалять или добавлять данные, определять права доступа, т. е. проделывать практически все допустимые для таблиц операции.

Когда выполняется запрос к представлению, СУБД выполняет следующие операции:

– выбирает определение представления из словаря данных;

– проверяет привилегии доступа;

– преобразует запрос, заданный в описании представления, в соответствующую операцию к таблице или таблицам БД.

Другими словами, данные выбираются или заносятся через представление в реальные таблицы БД.

Преимущества использования представлений. В локальных БД представления применяются для удобства и позволяют упрощать запросы. В промышленных БД представления играют главную роль в создании собственной структуры БД для каждого пользователя и обеспечении ее безопасности. Ниже перечислены основные преимущества представлений.

– Безопасность. Каждому пользователю можно разрешить доступ к небольшому числу представлений, содержащих только ту информацию, которую ему позволено знать (примеры горизонтальных и вертикальных представлений). Таким образом, можно осуществить ограничение доступа пользователей к хранимой информации.

– Простота запросов. С помощью представления можно извлечь данные из нескольких таблиц и представить их как одну таблицу, превращая тем самым запрос

ко многим таблицам в однотабличный запрос к представлению (такие представления также называют объединенными).

– Простота структуры. С помощью представлений для каждого пользователя можно создать собственную структуру БД, определив ее как множество доступных пользователю виртуальных таблиц.

– Защита от изменений. Представление может возвращать непротиворечивый и неизменный образ структуры БД, даже если исходные таблицы разделяются, реструктуризируются или переименовываются.

– Целостность данных. Если доступ к данным или ввод данных (возможно, не для всех представлений) осуществляется с помощью представления, СУБД может автоматически проверять, выполняются ли определенные условия целостности.

Недостатки представлений. Наряду с перечисленными выше преимуществами представления обладают и двумя существенными недостатками:

– Низкая производительность. Представление создает лишь видимость существования соответствующей таблицы, и СУБД приходится преобразовывать запрос к представлению в запрос к исходным таблицам. Если представление отображает многотабличный запрос, то простой запрос к представлению становится сложным объединением и на его выполнение может потребоваться много времени.

– Ограничения на обновление. Когда пользователь пытается обновить строки представления, СУБД должна установить их соответствие строкам исходных таблиц и обновить последние. Это возможно только для простых представлений; сложные представления так обновлять нельзя, они доступны только для выборки, но могут быть обновлены при помощи триггеров INSTEAD OF.

Указанные недостатки означают, что не стоит без разбора применять представления вместо исходных таблиц. В каждом конкретном случае необходимо учитывать перечисленные преимущества и недостатки.

Приведем примеры инструкций, создающих различные виды представлений. Допустим, необходимо создать представление, показывающее информацию о сотрудниках с заработной платой более 1000:

```
CREATE OR REPLACE VIEW Minsk AS
SELECT * FROM staff WHERE salary>1000;
```

Это пример создания простого *горизонтального представления*, которое позволяет видеть в исходной таблице *staff* не все строки, а только те, которые удовлетворяют условию отбора запроса, лежащего в его основе.

Можно создать представление, показывающее информацию о Ф. И. О. и должности служащих.

```
CREATE OR REPLACE VIEW info AS
SELECT fname, lname, position FROM staff;
```

Это пример создания простого *вертикального представления*, ограничивающего доступ к столбцам исходной таблицы.

Конечно, на практике при создании представлений обычно требуется разделять таблицу и по вертикали, и по горизонтали – такие представления получили название смешанных.

Также необходимо выделить сложные *сгруппированные представления*. В отличие от горизонтальных, вертикальных и смешанных представлений, каждой строке сгруппированного представления не соответствует какая-то одна строка исходной таблицы. Сгруппированное представление не является просто фильтром исходной таблицы, скрывающим некоторые строки и столбцы. Оно отображает исходную таблицу в виде резюме, поэтому поддержка такой виртуальной таблицы требует от СУБД значительного объема вычислений и отслеживания проблем с обновлением.

Создадим представление с информацией о средней заработной плате сотрудников по каждому отделению:

```
CREATE OR REPLACE VIEW average_salary as
SELECT bno, AVG (salary) Srednya_zarplata
FROM staff
GROUP BY bno;
```

Обновление представлений. Для простых представлений операции добавления, изменения и удаления можно преобразовать в эквивалентные операции по отношению к исходным таблицам. Например, добавление строки в представление *Minsk* имеет смысл и означает, что новая строка должна быть вставлена в таблицу *staff*, лежащую в основе представления. Аналогично имеют смысл обновление и удаление строки из представления *Minsk*. Во всех случаях требуемое действие можно выполнить по отношению к соответствующей строке исходной таблицы, тем самым сохраняя целостность исходной таблицы и представления.

Согласно стандарту представление можно обновлять в том случае, если определяющий его запрос соответствует перечисленным ниже требованиям.

1. Обязательно должен присутствовать первичный ключ и все столбцы, которые определены как NOT NULL.
2. Должен отсутствовать предикат DISTINCT, т. е. повторяющиеся строки не должны исключаться из результата запроса.
3. В предложении FROM должна быть задана только одна таблица или представление.
4. Каждое имя в списке возвращаемых столбцов должно быть ссылкой на простой столбец, не должны содержаться выражения, вычисляемые столбцы и статистические функции, но обязательно должен присутствовать ключевой столбец и столбцы, заданные с ограничением NOT NULL при создании таблицы.
5. В предложении WHERE не должен содержаться подчиненный запрос.
6. В запросе не должны присутствовать предложения GROUP BY и HAVING.

Обобщая сказанное, можно отметить, что если между строками представления и строками исходной таблицы есть соответствие «один к одному», то такое представление можно считать обновляемым. Если между строками представления и исходной таблицы нет однозначного соответствия, то добавление, удаление и изменение строк представления не имеют смысла и поэтому запрещены. Однако некоторые коммерческие СУБД позволяют обновлять «неоднозначные» представления. Так в *Oracle* такие обновления возможны посредством триггеров INSTEAD OF.

В *Oracle* инструкция создания представления имеет вид

CREATE OR REPLACE [FORCE|NOFORCE] VIEW *имя_представления* [(*столбец_представления* [,*столбец_представления* ...])] AS *SQL-запрос* [WITH CHECK OPTION [CONSTRAINT *имя_ограничения*]] [WITH READ ONLY];

где OR REPLACE – пересоздает представление, если оно уже существует. Можно использовать эту опцию для изменения определения представления без того, чтобы удалять его, создавать заново и вновь назначать все объектные привилегии, которые были назначены по данному представлению;

FORCE – создает представление независимо от того, существуют ли базовые таблицы этого представления, и от того, имеет ли владелец схемы, содержащей представление, привилегии по этим таблицам. Необходимо, чтобы оба названных условия были удовлетворены, прежде чем по данному представлению можно будет выдавать любые предложения SELECT, INSERT, UPDATE или DELETE. По умолчанию применяется параметр NOFORCE;

NOFORCE – создает представление только в том случае, если существуют базовые таблицы этого представления, а владелец схемы, содержащей представление, имеет привилегии по этим таблицам;

WITH CHECK OPTION – указывает, что вставки и обновления, которые будут осуществляться через этот запрос, должны давать в результате только такие строки, которые могут быть выбраны запросом этого же представления. Другими словами, если представление создается посредством запроса с предложением WHERE, то в представлении будут видны только строки, удовлетворяющие условию отбора. Остальные строки в исходной таблице присутствуют, но в представлении их нет.

Пусть существует горизонтальное представление *Minsk*, которое содержит строки таблицы *staff* с определенными значениями в столбце *bno*, которые соответствуют офисам компании, расположенным в Минске (пусть, например, это будет только офис с *bno* = 1).

Это представление является обновляемым как по стандарту *ANSI/ISO*, так и в *Oracle*, следовательно, в него можно добавить информацию о новом служащем посредством инструкции INSERT INTO:

```
INSERT INTO Minsk (sno, fname, lname, address, position, sex, dob, salary, bno)
VALUES ('s129', '…', '…', '…', 'менеджер', 'f', TO_DATE ('01.01.81', 'dd.mm.rr'), 300, 1).
```

СУБД добавит новую строку в исходную таблицу *staff*; она будет видна также в представлении *Minsk*. Кроме того, без опции WITH CHECK OPTION ничего не препятствует выполнению следующей инструкции:

```
INSERT INTO Minsk (sno, fname, lname, address, position, sex, dob, salary, bno)
VALUES ('s130', '…', '…', '…', 'менеджер', 'm', TO_DATE ('01.01.91', 'dd.mm.rr'), 300, 2).
```

После этого в запросе к представлению добавленная строка будет отсутствовать. Тот факт, что в результате выполнения инструкции INSERT или UPDATE из представления исчезают строки, в лучшем случае вызывает замешательство.

При указании параметра WITH CHECK OPTION пользователь не сможет вводить, удалять и обновлять информацию таблицы, из которой он не имеет возможности считать информацию через представление. Обновляемое представление, использующее несколько связанных таблиц, по стандарту не рекомендуется создавать с данным параметром, так как опция CHECK OPTION не может гарантировать контроль, если существует подзапрос в запросе этого представления или любого представления, на котором базируется данное представление, однако в СУБД *Oracle* это возможно.

CONSTRAINT *имя_ограничения* – задает имя, которое присваивается ограничению CHECK OPTION. Если этот идентификатор опущен, то *Oracle* автоматически назначает этому ограничению уникальное имя.

Когда для представления установлен режим контроля, СУБД автоматически проверяет каждую операцию INSERT или UPDATE, выполняемую над представлением, чтобы удостовериться в том, что полученные в результате строки удовлетворяют условиям отбора в определении представления. Если добавляемая или обновляемая строка не удовлетворяет этим условиям, то выполнение инструкции INSERT или UPDATE завершается ошибкой; другими словами, операция не выполняется.

WITH READ ONLY – явно запрещает все операции *DML* над представлением любого вида. Если команды обновления *DML* (INSERT, UPDATE, DELETE) можно применить к представлению, то говорят, что представление является обновляемым (*updatable*); в противном случае оно является только читаемым (*read-only*).

В *Oracle* представление является обновляемым, если при его создании учитывались следующие критерии:

– оно должно включать первичный ключ таблицы;

– не должно содержать полей, полученных в результате применения функций агрегирования;

– не должно содержать DISTINCT, GROUP BY, HAVING в своем определении;

– может быть определено на другом представлении, но это представление должно быть обновляемым;

– не может содержать константы, строки или выражения (например, comm*100) в списке выбираемых выходных полей.

Удаляются представления посредством инструкции DROP VIEW. Так, представление Minsk может быть удалено при выполнении инструкции

DROP VIEW Minsk;

Команда удаляет определение представления из БД. Поскольку строки и столбцы хранятся в таблицах БД, удаление представления на данные таблиц не влияет. Представления или другие приложения, основывающиеся на удаленных представлениях, становятся некорректными.

2.8. Псевдонимы таблиц (инструкции CREATE/DROP SYNONYM)

БД часто бывают организованы так, что все основные таблицы собраны вместе и принадлежат администратору. Администратор БД дает другим пользователям разрешения на доступ к таблицам, руководствуясь правилами обеспечения безопасности. Поэтому только в случае получения разрешения на доступ к таблицам другого пользователя для ссылки на них необходимо использовать полные имена таблиц. На практике это означает, что в каждом запросе к таким таблицам следует указывать полные имена таблиц, в результате чего запросы становятся длинными, а их ввод – утомительным.

Для решения этой проблемы во многих СУБД вводится понятие *псевдонима* или *синонима*. *Псевдоним* – это назначаемое пользователем имя, которое заменяет имя некоторой таблицы, представления, процедуры, функции или другого объекта схемы и наследует все их привилегии.

В *Oracle* для создания псевдонимов используется инструкция CREATE SYNONYM. В других СУБД и по стандарту для аналогичных целей используется инструкция CREATE ALIAS.

CREATE [PUBLIC] SYNONYM *имя_синонима*
FOR [*схема*.] *имя_объекта_схемы*[*@связь_БД*]

После создания псевдонима, например, для полного имени таблицы его можно использовать в запросах *SQL* как обычное имя таблицы. Применение псевдонимов смысл запроса не изменяет, так как и в этом случае необходимо иметь разрешение на доступ к таблицам других пользователей. Тем не менее псевдонимы упрощают инструкции *SQL*, и последние приобретают такой вид, как если бы вы обращались к своим собственным таблицам. Если псевдоним не нужен, то можно его удалить посредством инструкции

DROP SYNONYM *имя_синонима*.

2.9. Индексы (инструкции CREATE/DROP INDEX)

Одним из структурных элементов физической памяти, присутствующим в большинстве реляционных СУБД, является индекс. *Индекс* – это средство, обеспечивающее быстрый доступ к строкам таблицы на основе значений одного или нескольких столбцов. СУБД пользуется индексом так же, как читатели пользуются предметным указателем книги. В индексе хранятся значения данных и указатели на строки, где эти данные встречаются. Данные в индексе располагаются в отсортированном по убыванию или возрастанию порядке, чтобы СУБД могла быстро найти требуемое значение. Затем по указателю СУБД может быстро локализовать строку, содержащую искомое значение.

Наличие или отсутствие индекса совершенно незаметно для пользователя, обращающегося к таблице. Например, в случае поиска сотрудника по фамилии будет использоваться столбец *lname*. Если бы индекса для столбца не существовало, то СУБД была бы вынуждена выполнять запрос путем последовательного «сканирования» таблицы *staff*, строки за строкой, просматривая в каждой значение столбца *lname*. Для получения гарантии того, что найдены все строки, удовлетворяющие условию отбора, СУБД обязана просмотреть каждую строку таблицы. Если таблица имеет сотни тысячи строк, то ее просмотр может занять достаточно много времени.

Если для столбца *lname* имеется индекс, СУБД находит требуемые данные, просматривая индекс, чтобы найти требуемое значение, а затем с помощью указателя находит требуемую строку (строки) таблицы. Поиск в индексе осуществляется достаточно быстро, так как индекс отсортирован (сортировка символьных столбцов осуществляется в лексикографическом порядке) и его строки достаточно коротки. Переход от индекса к строке (строкам) также происходит довольно быстро, поскольку в индексе содержится информация о том, где на диске располагается эта строка (строки).

Как видно из примера, индекс имеет то преимущество, что ускоряет выполнение инструкций с условиями отбора, имеющими ссылки на индексный столбец (столбцы). К недостаткам индекса относится то, что, во-первых, он занимает на диске дополнительную память и, во-вторых, индекс необходимо обновлять каждый раз, когда в таблицу добавляется строка или обновляется индексный столбец таблицы. Это требует дополнительных затрат на выполнение инструкций INSERT и UPDATE, которые обращаются к данной таблице.

В общем-то, полезно создавать индекс лишь для тех столбцов, которые часто используются в условиях отбора. Индексы удобны также в тех случаях, когда инструкция SELECT обращается к таблице гораздо чаще, чем инструкции INSERT и UPDATE. СУБД всегда создает индекс для первичного ключа таблицы, так как ожидает, что доступ к таблице чаще всего будет осуществляться через первичный ключ.

В стандартах *SQL* ничего не говорится об индексах и о том, как их создавать. Они относятся к «деталям реализации», выходящим за рамки ядра языка

SQL. Тем не менее индексы весьма важны для обеспечения требуемой производительности любой серьезной, корпоративной или промышленной БД.

На практике в большинстве популярных СУБД (включая *Oracle*, *SQLServer*, *Informix, Sybase*) для создания индекса используется та или иная форма инструкции CREATE INDEX. В инструкции указывается имя индекса и таблица, для которой он создается, индексируемый столбец и порядок его сортировки (по возрастанию или убыванию). Наиболее подходящими для индексации полями являются: поля внешних ключей; поля, наиболее часто используемые в WHERE, GROUP BY, ORDER BY.

В *Oracle* можно создать, изменить или удалить индекс для одного или нескольких столбцов таблицы, используя следующий синтаксис:

CREATE [OR REPLACE] [UNIQUE | BITMAP] INDEX [*схема.*] *имя_индекса* ON [*схема*.]*имя_таблицы* [псевдоним] (*столбец* | *выражение_для_столбца* [ASC | DESC][, …]);

где UNIQUE – означает, что значения столбцов, на которые ссылается индекс, должны быть уникальными; BITMAP – изменение структуры индекса со сбалансированного дерева на структуру растровой карты.

В следующем примере создадим индекс с именем *lname_idx* для столбца таблицы *staff*:

```
CREATE INDEX lname_idx ON staff (lname)
   TABLESPACE all_oracle_tbs
 STORAGE (INITIAL 20K
        NEXT 20K
        PCTINCREASE 75
        PCTFREE 0);
```

Обратите внимание, что для индекса явно указано несколько настроек хранения в табличном пространстве в некотором гипотетическом пространстве *all_oracle_tbs*. Если для индекса не указывать параметры хранения, то будут использоваться параметры хранения, указанные при создании табличного пространства.

Индексы могут быть уникальными или неуникальными. Уникальные индексы гарантируют отсутствие двух строк в таблице с одинаковыми значениями в ключевом столбце или столбцах. Неуникальные индексы не накладывают никаких ограничений на значения столбцов.

Для создания уникального индекса используется оператор CREATE UNIQUE INDEX, что показано в следующем абстрактном примере:

```
CREATE UNIQUE INDEX anikname_unique_idx
ON all_oracle_admin (anikname) TABLESPACE all_oracle_idx;
```

Вместо создания уникальных индексов для нужных столбцов можно определить ограничение целостности UNIQUE. В этом случае *Oracle* обеспечивает

выполнение ограничений целостности UNIQUE, автоматически создавая уникальный индекс для уникального ключа.

Создание индексов, связанных с ограничением целостности. *Oracle* обеспечивает выполнение ограничения целостности UNIQUE или PRIMARY KEY для таблицы, создавая уникальный индекс для уникального или первичного ключа. Этот индекс создается автоматически, когда включается ограничение целостности. Когда выполняется CREATE TABLE или ALTER TABLE, для создания индекса не надо предпринимать никаких действий, но при желании можно указать предложение USING INDEX, чтобы контролировать его создание.

Чтобы включить ограничение целостности UNIQUE или PRIMARY KEY, создавая таким образом связанный с ним индекс, владелец таблицы должен иметь квоту табличного пространства, где будет храниться этот индекс, или системную привилегию UNLIMITED TABLESPACE. Индекс, связанный с ограничением целостности, всегда получает имя этого ограничения, если вы не укажете иное.

Параметры хранения для индексов, связанных с ограничением целостности UNIQUE или PRIMARY KEY, можно устанавливать с помощью предложения USING INDEX.

2.10. Последовательности (инструкции CREATE/DROP SEQUENCE)

Использование последовательностей полезно для автоматической генерации уникальных первичных ключей для данных, а также для координирования ключей между различными строками или таблицами.

Без генератора последовательностей порядковые номера можно создавать лишь программным способом. Новое значение первичного ключа можно было бы получать выбором последнего программно вычисленного значения и наращиванием его. Этот метод требует блокировки во время выполнения транзакции и заставляет одновременно работающих пользователей ожидать очередного значения первичного ключа; такое ожидание известно как СЕРИАЛИЗАЦИЯ (буквально – «выстраивание в очередь»). Если есть в пользовательских приложениях такие программные конструкции, то их нужно заменять на обращения к последовательностям. Последовательности устраняют сериализацию и улучшают конкурентные способности приложений.

Рассмотрим, как создавать, изменять и удалять последовательности с помощью команд *SQL*.

Общий синтаксис создания:

CREATE SEQUENCE [*схема*].*имя_последовательности*
[INCREMENT BY *шаг_последовательности*]
[START WITH *начало_последовательности*]
[MAXVALUE *максимальное_значение* | NOMAXVALUE]
[MINVALUE *минимальное_значение* | NOMINVALUE]
[CYCLE | NOCYCLE]

[CACHE *кешируемое_значение* | NOCACHE]
[ORDER | NOORDER];

где *имя_последовательности* – это уникальное имя последовательности;

шаг_последовательности – это шаг последовательности, по умолчанию это 1. Абсолютное значение этого параметра должно быть меньше, чем разница между конечным и начальным значениями;

начало_последовательности – это целочисленное значение, с которого начинается отсчет, по умолчанию это 1;

максимальное_значение – это максимальное значение последовательности; значение *maximum_num* должно быть больше или равно значению *start_num* и больше, чем значение *minimum_num*;

NOMAXVALUE – устанавливает максимальное значение равным 10^{27} для возрастающей последовательности или –1 для убывающей. Используется по умолчанию;

минимальное_значение – это минимальное значение последовательности; *минимальное_значение* должно быть меньше либо равно *начало_последовательности* и меньше, чем *минимальное_значение*;

NOMINVALUE – определяет минимальное значение, равное 1, для возрастающей последовательности и -10^{26} для убывающей. Используется по умолчанию;

CYCLE – подразумевает, что последовательность начинает генерировать значения по кругу при достижении максимального или минимального значения. При обращении к последовательности, когда она достигла максимального значения, следующее сгенерированное значение будет минимальным значением последовательности. В ситуации с убывающей последовательностью, при достижении минимального значения, следующее сгенерированное будет максимальным;

NOCYCLE – указывает прекратить генерацию значений при достижении максимума или минимума последовательности. Используется по умолчанию;

кешируемое значение – это количество значений, сохраняемых в памяти. По умолчанию это 20. Минимальное количество кешированных значений – 2, максимальное значение высчитывается по формуле

CEIL(*максимальное_значение – минимальное_значение*)/ABS(*шаг_последовательности*);

NOCACHE – отключает кеширование. Это не дает возможности БД выделить некоторое количество значений заблаговременно, что позволит избежать пробелов в последовательности, но увеличит затраты системных ресурсов. Пробелы могут возникнуть при остановке сервера БД, когда кешированные значения теряются. Если опции CACHE и NOCACHE не указываются, то по умолчанию кешируется 20 значений;

ORDER – обеспечивает генерацию значений в порядке запросов. Как правило, ORDER используется в среде *Real Application Clusters*. NOORDER – не дает таких гарантий. По умолчанию используется NOORDER.

Например, следующая команда создает последовательность, которую можно использовать для генерации номеров сотрудников для столбца *sno* таблицы *staff*:

```
CREATE SEQUENCE sno_sequence INCREMENT BY 1
   START WITH 1
   MAXVALUE 99999
   MINVALUE 1
   NOCYCLE
   CACHE 10;
```

Чтобы изменить последовательность (за исключением начального номера), используется команда ALTER SEQUENCE. Например, следующее предложение изменяет последовательность:

```
ALTER SEQUENCE sno_sequence
   INCREMENT BY 10
   MAXVALUE 10000
   CYCLE
   CACHE 20;
```

Чтобы изменить начальную точку последовательности, надо удалить эту последовательность и заново создать ее и все необходимые привилегии.

Однажды определенную последовательность можно сделать доступной многим пользователям, которые могут обращаться к этой последовательности и вызывать ее приращения, не прибегая к ожиданию. *Oracle* не ждет завершения транзакции, выполнившей приращение последовательности, для того чтобы осуществить очередное приращение этой последовательности.

Обращение к последовательности осуществляется в предложениях *SQL* через псевдостолбцы NEXTVAL и CURRVAL; каждый новый номер данной последовательности генерируется обращением к ее псевдостолбцу NEXTVAL, тогда как текущий номер последовательности можно извлекать неоднократно путем обращения к ее псевдостолбцу CURRVAL.

NEXTVAL и CURRVAL не являются зарезервированными или ключевыми словами; их можно использовать как имена псевдостолбцов в предложениях *SQL*, таких как SELECT, INSERT или UPDATE.

Чтобы сгенерировать и возвратить очередной номер последовательности, необходимо обратиться к *имя_последовательности*.NEXTVAL.

Самым простым примером может быть выполнение такого запроса:

```
SELECT sno_sequence.NEXTVAL FROM DUAL;
```

Или попробуем использовать ее для генерации значений первичного ключа при вставке данных в таблицу:

```
INSERT INTO staff (bno, address, tel_no)
VALUES (sno_sequence.NEXTVAL, 'Пушкина 12 ', '313-23-23');
```

или во фразе SET предложения UPDATE, например:

```
UPDATE staff
SET bno = sno_sequence.NEXTVAL WHERE bno=10112;
```

По определению первое обращение к *sno_sequence*.NEXTVAL возвратит значение 1. Каждое последующее предложение, обращающееся к *sno_sequence*.NEXTVAL, возвратит очередной номер данной последовательности (2, 3, 4 и т. д.). Псевдостолбец NEXTVAL может генерировать столько новых номеров последовательности, сколько потребуется. Однако на одно предложение генерируется лишь один новый номер; иными словами, если в данном предложении *SQL* псевдостолбец NEXTVAL встречается несколько раз, то лишь для первого обращения будет возвращен новый номер последовательности, а все остальные обращения в этом предложении возвратят тот же самый номер. После того как очередной номер последовательности сгенерирован, этот номер доступен лишь сессии, сгенерировавшей его. Независимо от подтверждения или отката транзакций все пользователи, обращающиеся к *sno_sequence*.NEXTVAL, получают уникальные значения. Поэтому, если несколько пользователей одновременно обращаются к одной и той же последовательности, каждый из них может получать номера этой последовательности с промежутками, потому что номера генерируются также другими пользователями.

Чтобы обратиться к текущему значению номера последовательности, которое уже было сгенерировано для вашей сессии, используйте обозначение *имя_последовательности*.CURRVAL. Псевдостолбец CURRVAL может использоваться лишь в том случае, если в текущей сессии уже было выдано обращение к *имя_последовательности*.NEXTVAL для данной последовательности (не обязательно в текущей транзакции). CURRVAL можно использовать сколько угодно раз, в том числе несколько раз в одном и том же предложении. Очередной номер последовательности не будет сгенерирован, пока не будет выполнено очередное обращение к NEXTVAL.

NEXTVAL и CURRVAL могут использоваться во фразе VALUES предложения INSERT, в списке SELECT предложения SELECT, во фразе SET предложения UPDATE.

NEXTVAL и CURRVAL не могут использоваться в следующих местах:

– в подзапросе;

– запросе, определяющем представление или моментальный снимок;

– предложении SELECT с оператором DISTINCT;

– предложении SELECT с фразой GROUP BY или ORDER BY;

– предложении SELECT, скомбинированном с другим предложением SELECT одним из операторов множеств UNION, INTERSECT или MINUS;

– во фразе WHERE предложения SELECT;

– в выражении DEFAULT для столбца в предложении CREATE TABLE или ALTER TABLE;

– в условии ограничения CHECK.

Если последовательность больше не нужна, можно удалить ее с помощью команды *SQL* DROP SEQUENCE. Например, следующее предложение удаляет последовательность *order_seq*:

DROP SEQUENCE *order_seq*;

При удалении последовательности ее определение удаляется из словаря данных. Все синонимы для последовательности остаются, но возвращают ошибку при обращении к ним.

Контрольные вопросы и задания к разделу 2

1) Создайте в *SQL*Plus* или в облачной версии *iSQL*Plus* модель учебной БД малого предприятия по аренде недвижимости (рис. 2.3), которая и в дальнейшем будет использоваться для пояснения работы инструкций *SQL*. БД содержит шесть таблиц: *branch*, *staff*, *property_for_rent*, *renter*, *owner*, *viewing*.

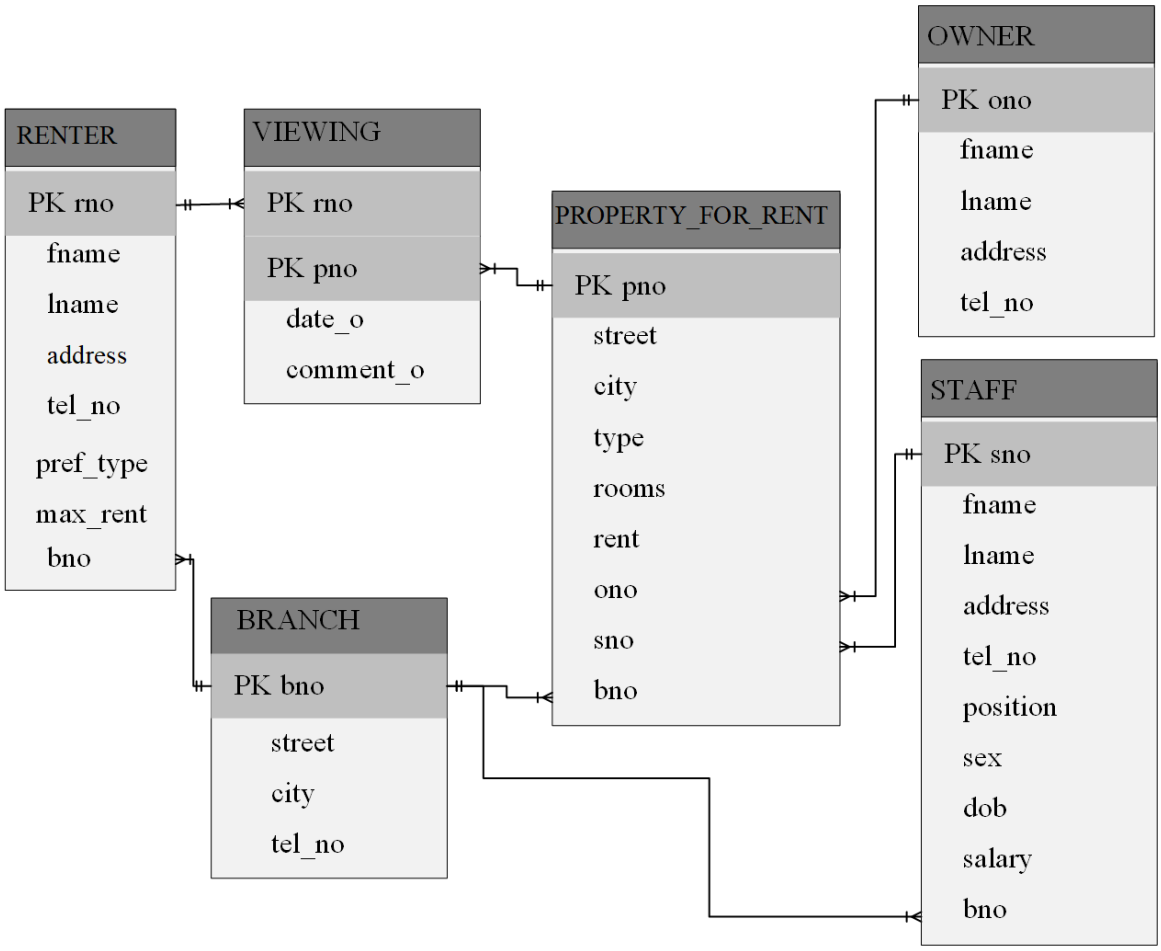


Рис. 2.3 Схема данных агентства недвижимости

Таблица *branch* предназначена для хранения информации об отделениях (офисах) предприятия и имеет следующие атрибуты: *bno*, *street*, *city*, *tel_no*, где атрибут *bno* является первичным ключом.

Таблица *staff* предназначена для хранения информации о сотрудниках и содержит следующие атрибуты: *sno, fname, lname, address, tel_no, position, sex, dob, salary, bno*. *Sno* – первичный ключ, идентификатор записей о сотрудниках; *position* – строковый атрибут, определяет занимаемую должность; *dob* – атрибут типа дата/время с данными о днях рождения сотрудников; *salary* – числовой атрибут с данными о заработной плате сотрудников. Атрибут *bno* – внешний ключ для связи с таблицей *branch*.

Таблица *property_for_rent* содержит информацию об объектах недвижимости, предлагаемых в аренду, и имеет следующие атрибуты: *pno, street, city, type, rooms, rent, ono, sno, bno*, где *pno* – первичный ключ, *type* – строковый атрибут с информацией о типе предлагаемого объекта недвижимости; в данном случае на значения атрибута наложено ограничение, он может принимать либо значение «*h*» – дом (*house*), либо «*f*» – квартира (*flat*). *Rooms* и *rent* – числовые атрибуты. *Ono, sno, bno* – внешние ключи таблицы для связи с таблицами *owner, staff, branch* соответственно, причем наличие связи с таблицей *branch* в данном случае означает, что объект может быть закреплен за одним отделением, а обслуживаться сотрудником из другого.

Таблица *renter* содержит информацию о потенциальных арендаторах и содержит следующие атрибуты: *rno, fname, lname, address, tel_no, pref_type, max_rent, bno*. *Rno* – первичный ключ, *pref_type* – строковый атрибут, определяющий предпочтительный для клиента тип объекта аренды и ограниченный значениями «*h*» и «*f*». *Max_rent* – числовой атрибут, имеющий смысл максимальной рентной стоимости объекта с точки зрения арендатора, *bno* – внешний ключ для связи с таблицей *branch*.

Таблица *owner* определяет владельцев объектов недвижимости, которые сдаются в аренду и включает следующие поля: *ono, fname, lname, address, tel_no*.

Таблица *viewing* содержит результаты осмотра арендаторами объектов аренды и имеет поля: *rno, pno, date_o, comment_o*. Таблица имеет составной первичный ключ, включающий атрибуты *rno* и *pno*, каждый из которых является также внешним ключом для связи с таблицами *renter* (указывает, кто из арендаторов производил осмотр) и *property_for_rent* (какой из объектов осматривался). Атрибут *date_o* определяет дату осмотра, а *comment_o* предназначен для сохранения сделанных потенциальным арендатором комментариев.

2) Дополнительно создайте таблицу position, установите связь «один ко многим» с таблицей staff.

3) Добавьте в таблицу branch столбец для хранения информации о руководителе отдела, установите связь с таблицей staff.

4) Создайте представление Minsk для хранения и работы с информацией только о сотрудниках, работающих в минских отделениях.

5) Создайте индексы для всех внешних ключей таблиц.

6) Создайте последовательность, которая начинается с 10, имеет шаг 10 и не кеширует значения.

7) В каких случаях СУБД создает индексы автоматически?

8) Для чего и как используют синонимы и последовательности в БД?

3. ОПЕРАТОРЫ ЯЗЫКА МАНИПУЛИРОВАНИЯ ДАННЫМИ

3.1. Общие сведения об инструкциях *DML*

К *DML*-инструкциям в *Oracle* относятся команды вставки, обновления, удаления, чтения, изменения способа работы сервера при обращении к данным.

Перечислим основные инструкции *DML* в табл. 3.1.

Таблица 3.1

Основные инструкции *DML*

Инструкция	Назначение
ALTER SESSION	Изменяет функциональные характеристики текущего сеанса связи с БД (*+NLS*), многие из параметров определены в файле инициализации *INIT.ORA* или *SPFILE*
ANALYZE	Собирает или удаляет статистическую информацию об объекте БД, проверяет структуру объекта или идентифицирует переменные и сцепленные строки таблицы (кластера)
DELETE	Удаляет строки из таблицы, представления или моментальной копии
EXPAIN PLAN	Создает пояснение для плана использования (исполнения) команды *SQL*
INSERT INTO	Вставляет строки данных в таблицу или представление
MERGE INTO	Выбирает строки таблицы для обновления или вставки в другую таблицу (с *Oracle*9*i Release* 2) (позволяет избегать многократного использования INSERT, UPDATE)
SAVE POINT*	Определяет точку сохранения транзакции, до которой можно было выполнить откат при помощи команды ROLLBACK
SELECT	Извлекает данные из таблиц, представлений и моментальных копий
SET CONSTRAINT	Задает на уровне транзакций, будут ли ограничения проверяться после каждой *DML*-команды или только в конце транзакций
SET ROLE	Включает или отключает роль для текущего сеанса
SET TRANSACTION*	Устанавливает базовые характеристики транзакции
TRUNCATE*	Удаляет все строки из таблицы или кластера (не создает записей отката, выполняется быстро, аналог DELETE FROM)
UPDATE	Изменяет значение, хранящееся в одном или нескольких столбцах данных, в одной или нескольких таблицах, представлениях или моментальных копиях

* В общепринятом смысле эти инструкции не относятся к *DML*, однако именно так они определены в источнике [5].

В стандарте *SQL* описаны только наиболее распространенные инструкции SELECT, INSERT INTO, UPDATE и DELETE, которые подробно рассмотрены далее, как и несколько других инструкций из табл. 3.1.

3.2. Инструкция SELECT (общий синтаксис)

Все запросы на получение практически любого количества данных из одной или нескольких таблиц выполняются с помощью предложения SELECT. В общем случае результатом реализации предложения SELECT является другая таблица. К этой новой таблице может быть снова применена операция SELECT и т. д., т. е. такие операции могут быть вложены друг в друга.

Предложение SELECT может использоваться как:

– самостоятельная команда на получение и вывод строк таблицы, сформированной из столбцов и строк одной или нескольких таблиц (представлений);

– элемент WHERE- или HAVING-условия («вложенный запрос»);

– фраза выбора в командах CREATE VIEW, DECLARE CURSOR или INSERT;

– средство присвоения глобальным переменным значений из строк сформированной таблицы (фраза INTO).

Инструкция SELECT – наиболее мощная из всех *SQL*-инструкций и, по сути, аналогична таким операторам реляционной алгебры, как *selection, projection, joining*.

Инструкция SELECT состоит из семи основных предложений.

Предложения SELECT и FROM являются обязательными. Четыре остальных включаются в инструкцию только при необходимости. Ниже перечислены функции каждого из предложений.

В предложении SELECT указывается список столбцов, которые должны быть возвращены инструкцией SELECT. Возвращаемые столбцы могут содержать значения, извлекаемые из столбцов таблиц БД или вычисляемые во время выполнения запроса.

В предложении FROM указывается список таблиц, которые содержат элементы данных, извлекаемые запросом.

Предложение WHERE показывает, что в результаты запроса следует включать только некоторые строки. Для отбора строк, включаемых в результаты запроса, используется условие отбора.

Предложение GROUP BY позволяет создать итоговый запрос. Обычный запрос включает в результаты запроса по одной записи для каждой строки из таблицы. Итоговый запрос, напротив, вначале группирует строки БД по определенному признаку, а затем включает в результаты запроса одну итоговую строку для каждой группы.

Предложение HAVING показывает, что в результаты запроса следует включать только некоторые из групп, созданных с помощью предложения GROUP BY. В этом предложении, как и в предложении WHERE, для отбора включаемых групп используется условие отбора.

Предложение ORDER BY сортирует результаты запроса на основании данных, содержащихся в одном или нескольких столбцах. Если это предложение не указано, результаты запроса не будут отсортированы.

Предложение FETCH позволяет ограничить количество строк, выводимых запросом.

Порядок выполнения этих команд в обычном запросе следующий: FROM, WHERE, GROUP BY, HAVING, SELECT, ORDER BY, FETCH.

Полный формат инструкции SELECT, применяемой в *Oracle*, имеет вид [5]:

SELECT [{DISTINCT | UNIQUE | ALL}]
{[*схема*.] [*имя_таблицы*. | *представление*. | *моментальная_копия*.]
имя_столбца[[AS] *псевдоним*] [, *имя_столбца* [[AS] *псевдоним*]…]
| *выражение* [[AS] *псевдоним*] [, *выражение* [[AS] *псевдоним*]…] | *}
FROM {*имя_таблицы* [@*связь*_БД] [AS OF {SCN | TIMESTAMP} *выражение*]
| *имя_таблицы* PARTITION (*имя_раздела*) [AS OF {SCN | TIMESTAMP} *выражение*]
| *имя_таблицы* SUBPARTITION (*имя_подраздела*) [AS OF{SCN | TIMESTAMP}
выражение]
| *имя_таблицы* SAMPLE[BLOCK] *процент_выборки* [AS OF
{SCN | TIMESTAMP} *выражение*]
| *представление*[@*связь*_БД] [AS OF {SCN | TIMESTAMP} *выражение*]
| *представление* PARTITION (*имя_раздела*) [AS OF {SCN | TIMESTAMP} *выражение*]
| представление SUBPARTITION (имя_подраздела) [AS OF {SCN | TIMESTAMP} выражение]
| *представление* SAMPLE [BLOCK] *процент_выборки* [AS
OF {SCN | TIMESTAMP} *выражение*]
| *моментальная_копия*[@*связь*_БД] [AS OF{SCN | TIMESTAMP} *выражение*]
| *моментальная_копия* PARTITION (*имя_раздела*) [AS OF {SCN | TIMESTAMP} *выражение*]
| *моментальная_копия* SUBPARTITION (*имя_подраздела*) [AS OF
{SCN | TIMESTAMP} *выражение*]
| *моментальная_копия* SAMPLE [BLOCK] *процент_выборки* [AS OF
{SCN | TIMESTAMP} *выражение*]
| (*подзапрос*) | *имя_таблицы* {[*тип_соединения*] JOIN *имя_таблицы* { ON *условие*
|USING (*столбец* [, *столбец*]…)
|{CROSS JOIN|NATURAL [*тип_соединения*] JOIN *имя_таблицы*}
}
[, [*псевдоним_таблицы*]
[, *имя_таблицы* [@*связь*_БД] [AS OF {SCN | TIMESTAMP} *выражение*]
| *имя_таблицы* PARTITION (*имя_раздела*) [AS OF {SCN | TIMESTAMP} *выражение*]
| *имя_таблицы* SUBPARTITION (*имя_подраздела*) [AS OF{SCN | TIMESTAMP} *выражение*]
| *имя_таблицы* SAMPLE[BLOCK] *процент_выборки* [AS OF
{SCN | TIMESTAMP} *выражение*]
| *представление*[@*связь*_БД] [AS OF {SCN | TIMESTAMP} *выражение*]
| *моментальная_копия*[@*связь*_БД] [AS OF{SCN | TIMESTAMP} *выражение*]
| (*подзапрос*) | *имя_таблицы* {[*тип_соединения*] JOIN *имя_таблицы* { ON *условие*
|USING (*столбец* [, *столбец*]…) }

|{CROSS JOIN | NATURAL [*тип_соединения*] JOIN *имя_таблицы*}
} } [*псевдоним_таблицы*]]
[WHERE *условие*]
{[GROUP BY {*выражение* | {*выражение* [, *выражение* …]}
| CUBE (*выражение* [, *выражение* …]) | ROLLUP (*выражение* [, *выражение* …])
}
GROUPING SETS ({*выражение* | {*выражение* [, *выражение* …]}
|CUBE (*выражение* [, *выражение* …]) | ROLLUP (*выражение* [, *выражение* …])
]}
[HAVING *условие*]
[[START WITH *условие*] CONNECT BY *условие*]
[{UNION [ALL]] | INTERSECT | MINUS} (*подзапрос*)
[,{UNION [ALL] | INTERSECT | MINUS} (*подзапрос*)…]]
[ORDER BY{*выражение* | *позиция* | *псевдоним*} [ASC | DESC]
[, {*выражение* | *позиция* | *псевдоним*} [ASC | DESC] …]
[OFFSET *offset* ROWS]
[FETCH { FIRST | NEXT }[{*row_count* | *percent* PERCENT}] ROW | ROWS [ONLY | WITH TIES]]
FOR UPDATE[OF] { *имя_таблицы* | *представление*}. *столбец*
[{*имя_таблицы* | *представление*}. *столбец*] [NOWAIT]

При описании инструкции используются следующие ключевые слова:

– DISTINCT – указывает, что должна быть возвращена только одна копия строки, даже в случае наличия дубликатов строк;

– AS – задает псевдоним для столбца или выражения, может отсутствовать;

– ALL – указывает, что должны быть возвращены все строки, включаю дубликаты. По умолчанию ALL;

– AS OF – обеспечивает работу с данными по состоянию на конкретный номер системного изменения (SCN) или временную метку, как указывает *выражение*;

– SAMPLE [BLOCK] – выполняется случайная (или блочная) выборка строк таблицы;

– START WITH – указывает строки, выступающие в качестве корневых в иерархическом запросе (если эти ключевые слова пропущены, то все строки таблицы считаются корневыми);

– CONNECT BY – определяет отношение между родительскими и дочерними строками иерархии, направление обхода задается ключевым словом PRIOR;

– INNER – внутреннее соединение (установлено по умолчанию);

– RIGHT – правое внешнее соединение;

– LEFT – левое внешнее соединение;

– FULL – полное внешнее соединение;

– ON – задает *условие* соединения, которое не привязано к *условию* WHERE;

– USING – соединение по равенству для столбцов, имеющих одинаковые имена;

– CROSS JOIN – перекрестное соединение, т. е. прямое произведение двух отношений;

– NATURAL – естественное соединение по одноименным столбцам двух таблиц с одинаковыми значениями;

– JOIN – явное указание на соединение, эквивалентно перечислению таблиц через запятую;

– CUBE – группировка на основе всех возможных комбинаций значений предложенного списка отношений;

– ROLLUP – группировка на основе значений предложенного списка выражений и сводных строк, возвращенных для каждого выражения наряду с дополнительной строкой общего итога;

– GROUPING SETS – задает несколько групп данных для более удобного агрегирования (если указаны только нужные группы, то серверу не приходится выполнять все множество агрегатирований, которые требуют CUBE и ROLLUP);

– FOR UPDATE [OF] – означает, что выбранные строки будут заблокированы. Если указывается ключевое слово OF, то будут заблокированы только строки названной таблицы;

– NOWAIT – означает, что если таблица уже заблокирована, то сервер *Oracle* не будет ждать снятия блокировки.

3.3. Предложения SELECT и FROM

В предложении SELECT, с которого начинаются все инструкции SELECT, необходимо указать элементы данных, которые будут возвращены в результате запроса. Эти элементы задаются в самом простом случае в виде списка возвращаемых столбцов, разделенных запятыми. Для каждого элемента из этого списка в таблице результатов запроса будет создан один столбец. Столбцы в таблице результатов будут расположены в том же порядке, что и элементы списка возвращаемых столбцов. Возвращаемый столбец может представлять собой:

– имя столбца, идентифицирующее один из столбцов, содержащихся в таблицах, которые перечислены в предложении FROM. СУБД просто берет значение этого столбца для каждой из строк исходной таблицы и помещает его в соответствующую строку таблицы результатов запроса;

– константу, показывающую, что в каждой строке результатов запроса должно содержаться одно и то же значение;

– выражение, показывающее, что СУБД должна вычислять значение, помещаемое в таблицу результатов запроса, по формуле, определенной в выражении. Такие столбцы таблицы результатов носят название вычисляемых.

Предложение FROM состоит из ключевого слова FROM, за которым следует список спецификаторов таблиц, разделенных запятыми. Каждый спецификатор таблицы идентифицирует таблицу, содержащую данные, которые извлекает запрос. Такие таблицы называются исходными таблицами запроса.

Результатом *SQL*-запроса на выборку всегда является результирующий набор. Если пользователь выполняет инструкцию *SQL* в интерактивном режиме, СУБД выводит результаты запроса на экран в табличной форме. Если программа посылает запрос СУБД с помощью программного *SQL*, то СУБД возвращает результирующий набор программе.

Даже в случае, если запрос возвращает ноль строк, его результат считается таблицей. В этом случае таблица содержит заданное количество столбцов и ноль строк. Поддержка отсутствующих данных в *SQL* распространяется и на результаты запроса. Если один из элементов данных в исходной таблице имеет значение NULL, то оно попадет в результаты запроса при извлечении этого элемента.

То, что *SQL*-запрос всегда возвращает таблицу, означает следующее:

– результаты запроса можно записать обратно в БД в виде таблицы;

– результаты двух запросов, имеющие похожую структуру, можно объединить в одну таблицу;

– результаты запроса сами могут стать предметом дальнейших запросов.

Таким образом, табличная структура реляционной БД тесно связана с реляционными запросами *SQL*. Таблицам можно посылать запросы, а запросы возвращают таблицы.

Простые запросы. Простые запросы извлекают данные из столбцов, расположенных в одной таблице БД. Например, следующий запрос извлекает из таблицы *staff* три столбца:

```
SELECT fname, lname, position
FROM staff;
```

На логическом уровне запрос выполняется путем построчного просмотра таблицы, указанной в предложении FROM. Для каждой строки таблицы берутся значения столбцов, входящих в список возвращаемых столбцов, и создается одна строка результатов запроса. Таким образом, таблица результатов простого запроса на выборку содержит одну строку для каждой строки исходной таблицы БД.

Вычисляемые столбцы. Кроме столбцов, значения которых извлекаются непосредственно из БД, *SQL*-запрос на выборку может содержать вычисляемые столбцы, значения которых определяются на основании значений, хранящихся в БД. Чтобы получить вычисляемый столбец, в списке возвращаемых столбцов необходимо указать выражение. Выражения могут включать в себя любые арифметические операции, встроенные функции, скобки. Столбцы, участвующие в арифметическом выражении, должны содержать числовые данные. При попытке выполнения арифметических операций над столбцами, содержащими текстовые

данные, будет выдано сообщение об ошибке, так как к ним применима только операция **конкатенации**.

При выполнении следующего запроса на сотрудников с указанием зарплаты с 10 % надбавкой создается вычисляемый столбец:

```
SELECT fname, lname, position, (salary + 0.1*salary) AS percent
FROM staff;
```

Ключевое слово AS используют для того, чтобы присвоить новое имя вычисляемому столбцу или использовать сокращенное имя столбца вместо полного.

Фраза SELECT может включать не только выражения, но и отдельные числовые или текстовые константы. Текстовые константы должны заключаться в апострофы ('). Это необходимо для создания результата запроса в более удобном для пользователя виде.

```
SELECT pno, type, 'with rent', rent, 'per cent'
FROM property_for_rent;
```

В реальной работе очень часто приходится производить слияние строковых значений. Такая операция называется конкатенацией.

В *Oracle* в качестве оператора конкатенации используются две вертикальные черты (||) или встроенная функция CONCAT, которая работает на любых платформах.

Пример применения оператора конкатенации может выглядеть так:

```
SELECT fname || ' ' ||  lname  FIO  FROM staff;
```

Здесь сливаются три значения: имя, пробел (в виде литерала) и фамилия.

Оператор конкатенации можно применять как для строковых значений и значений CLOB (больших строковых значений), так и для чисел и дат (эти значения будут автоматически конвертированы в строковые). При слиянии строкового значения со значением типа NULL *Oracle* вернет строковое значение, однако такое поведение в будущих версиях *Oracle* не гарантируется. Поэтому в таких ситуациях рекомендуется использовать функцию NVL, которая преобразует значения типа NULL в строки нулевой длины.

При выполнении конкатенации *Oracle* использует для результата конкатенации больший тип данных из тех значений, которые в ней используются. Например, при слиянии двух значений с типом данных CHAR (максимальный размер – 2000 символов) вернется CHAR, при слиянии CHAR и VARCHAR2 (максимальный размер – 4000 символов) вернется VARCHAR2, а при слиянии любого типа данных с CLOB вернется CLOB.

Пробелы при конкатенации всегда сохраняются.

Выборка всех столбцов. Иногда требуется получить содержимое всех столбцов таблицы, например необходимо составить представление о структуре незнакомой базы и хранимых в ней данных. С учетом этого в *SQL* разрешается

использовать вместо списка возвращаемых столбцов символ «*», который означает, что требуется извлечь все столбцы в том порядке, в каком они расположены в исходной таблице:

```
SELECT * FROM staff;
```

В стандарте *ANSI*/*ISO* сказано, что в предложении SELECT может использоваться либо символ выборки всех столбцов, либо список возвращаемых столбцов, однако большинство коммерческих СУБД допускают использование и того и другого одновременно, например: SELECT *, salary FROM staff. В *Oracle*, как и требует стандарт, такой запрос не работает.

Повторяющиеся строки. Если в списке возвращаемых столбцов запроса на выборку указать первичный ключ таблицы, то каждая строка результатов запроса будет уникальной. Если первичный ключ не указан, результаты запроса могут содержать повторяющиеся строки. Например, необходимо определить, в каких городах существуют отделения предприятия:

```
SELECT DISTINCT city
FROM branch;
```

Если в одном и том же городе существует несколько отделений, название этого города будет повторяться в результирующей таблице.

Повторяющиеся строки из таблицы результатов запроса можно удалить, если в инструкции перед списком возвращаемых столбцов указать предикат DISTINCT.

DISTINCT может указываться только один раз в данном предложении SELECT. Если предложение выбирает многочисленные поля, DISTINCT опускает строки, где все выбранные поля идентичны. Строки, в которых некоторые значения одинаковы, а некоторые различны, будут сохранены. DISTINCT фактически приводит к показу всей строки вывода, не указывая полей (за исключением, когда он используется внутри агрегатных функций), так что нет никакого смысла, чтобы его повторять.

Вместо DISTINCT можно указать необязательное ключевое слово ALL, тогда дублирование строк вывода сохранится.

3.4. Сортировка и ограничение строк в результате запроса

Сортировка результатов запроса (предложение ORDER BY). Строки результатов запроса, как и строки таблицы БД, не имеют определенного порядка. Поэтому, включив в инструкцию SELECT предложение ORDER BY, можно отсортировать результаты запроса. При этом упорядочение можно производить в порядке возрастания – ASC (*ASCending*) или убывания DESC (*DESCending*), по умолчанию принимается ASC.

В предложение ORDER BY входит список имен, разделенных запятыми.

Пусть необходимо вывести отделения компании, отсортированные в алфавитном порядке по названию городов:

```
SELECT bno, city, tel_no
FROM branch
ORDER BY city;
```

Если столбец результатов запроса, используемый для сортировки, является вычисляемым, то у него нет имени, которое можно указать для сортировки. В таком случае вместо имени столбца необходимо указать его порядковый номер или алиас (псевдоним).

Если в списке ORDER BY несколько столбцов, то используется понятие старшего и младшего ключей сортировки. Особенно удобно использовать это в многотабличных запросах.

Пусть для каждого отделения компании нужно вывести Ф. И. О. работников, отвечающих за какие-либо объекты, а также перечень этих объектов в порядке убывания:

```
SELECT s.bno, s.sno, fname, lname, pno
FROM staff s, property_for_rent p
WHERE s.sno=p.sno
ORDER BY s.bno, s.sno, pno DESC;
```

Кроме того, необходимо знать, что *Oracle* при сортировке по возрастанию помещает NULL-значения в конец результирующего набора. Для того чтобы NULL-значения шли в начале результирующего набора при выполнении сортировки, нужно воспользоваться конструкцией NULLS FIRST. Также имеет место и конструкция NULLS LAST, но в ее использовании нет необходимости.

Ограничение количества выводимых данных. Начиная с версии *Oracle* 12*c* можно выполнить ограничение выводимых строк с помощью предложения FETCH в инструкции SELECT.

Синтаксис предложения следующий:

```
[OFFSET смещение ROWS]
FETCH { FIRST | NEXT }[ {количество_строк | процент PERCENT}] ROW | ROWS [ ONLY | WITH TIES ]
```

Ключевое слово OFFSET указывает количество пропускаемых строк перед началом вывода, является необязательным и если его нет, то смещение будет равно 0.

Смещение (*offset*) должно быть числом или выражением, которое оценивается как число. *Смещение* подчиняется следующим правилам:

– если смещение отрицательное, то оно рассматривается как 0;

– если смещение равно NULL или больше числа строк, возвращаемых запросом, строки не возвращаются;

– если смещение – дробь, то дробная часть усекается.

Предложение FETCH указывает *количество_строк* (*row_count*) или *процент* (*percent*) возвращаемых строк. В целях семантической ясности можно использовать ключевое слово ROW вместо ROWS и FIRST вместо NEXT, т. е. фразы FETCH NEXT 1 ROWS и FETCH FIRST 1 ROW по сути одинаковы. Если *количество_строк* или *процент* больше числа записей, то вернутся все строки.

Ключевое слово ONLY возвращает точное количество строк или процент строк.

Фраза WITH TIES возвращает дополнительные строки с тем же ключом сортировки, что и последняя выбранная строка, т. е. если следующее значение сортируемого поля равно последнему значению в окне отобранных. Если используется WITH TIES, то предложение ORDER BY обязательно, иначе запрос не будет возвращать дополнительные строки.

Например, выберем все строки, начиная с третьей:

```
SELECT * FROM staff
ORDER BY sno
OFFSET 2 ROWS;
```

Выберем только три строки, начиная с третьей:

```
SELECT * FROM staff
ORDER BY sno
OFFSET 2 ROWS
FETCH NEXT 3 ROWS ONLY;
```

Выберем пять сотрудников и их заработные платы:

```
SELECT  lname,  salary FROM  staff
ORDER BY  salary DESC
FETCH NEXT 5 ROWS WITH TIES;
```

Здесь опция WITH TIES приводит к тому, что запрос вернет больше пяти строк, если значение *salary* будет равно последнему отобранному и для других строк в таблице.

Следующий запрос вернет 5 % строк от отсортированной по убыванию заработных плат выборки:

```
SELECT lname, salary FROM staff
ORDER BY 2 DESC
FETCH FIRST 5 PERCENT ROWS ONLY;
```

Ранее в *Oracle* для ограничения количества выводимых данных в запросах использовался специальный псевдостолбец ROWNUM. В этот псевдостолбец сохраняется результат нумерации строк в результирующем наборе SELECT и использовать его можно было только в запросе вида

```
SELECT * FROM staff WHERE ROWNUM < 5;
```

В этом случае будет выведено всего четые записи. Такую возможность очень удобно использовать, когда запрос потенциально может возвращать очень большое количество значений и сильно загружать сервер, в то время как есть необходимость только убедиться в работоспособности кода. Неудобство работы с ROWNUM заключается в том, что при использовании условия больше (>) он всегда возвращает пустое множество, что требует написания дополнительных подзапросов.

3.5. Отбор строк (предложение WHERE)

Таблицы имеют тенденцию к увеличению своего размера, поскольку с течением времени все большее и большее количество строк в нее добавляется.

SQL-запросы, извлекающие из таблицы все строки, полезны при просмотре БД и создании отчетов, но редко применяются для чего-нибудь еще. Обычно требуется выбрать из таблицы несколько строк и включить в результаты запроса только их. Чтобы указать, какие строки нужно отобрать, следует воспользоваться предложением WHERE.

Предложение WHERE состоит из ключевого слова WHERE, за которым следует условие отбора, определяющее, какие именно строки требуется извлечь. Если условие отбора имеет значение TRUE, строка будет включена в результаты запроса. Если же оно имеет значение FALSE или NULL, то строка исключается из результатов запроса.

В *SQL* обычно используется пять основных условий отбора (в стандарте *ANSI*/*ISO* они называются предикатами):

– сравнение;

– проверка на принадлежность диапазону;

– проверка на членство во множестве;

– проверка на соответствие шаблону;

– проверка на значение NULL.

Операторы сравнения. Наиболее распространенным условием отбора в *SQL* является сравнение. При сравнении СУБД вычисляет и сравнивает значения двух выражений для каждой строки данных. В *SQL* имеется шесть различных способов сравнения выражений (по количеству операций сравнения). В *Oracle* операции сравнения обозначаются посредством следующих символов: =, != (не равно), <, >, <=, >=.

Приведем примеры использования запросов с условиями отбора типа сравнения. Необходимо найти служащих, родившихся до 1988 года:

```
SELECT fname, lname, dob
FROM staff
WHERE dob < to_date ('01.01.88','dd.mm.rr');
```

Вывести адреса и номера телефонов всех не минских отделений:

```
SELECT city, tel_no
FROM branch
WHERE city != 'Минск';
```

Условие сравнения можно делать и составным, объединяя простые сравнения логическими предикатами:

– AND – когда должны удовлетворяться оба разделяемых с помощью AND условия;

– OR – когда должно удовлетворяться одно из разделяемых с помощью OR условий;

– AND NOT – когда должно удовлетворяться первое условие и не должно второе;

– OR NOT – когда или должно удовлетворяться первое условие, или не должно удовлетворяться второе.

Существует приоритет AND над OR (сначала выполняются все операции AND и только после этого операции OR). Для получения желаемого результата WHERE условия должны быть введены в правильном порядке, который можно организовать введением скобок.

При обработке условия числа сравниваются алгебраически: отрицательные числа считаются меньшими, чем положительные, независимо от их абсолютной величины. Строки символов сравниваются в соответствии с их представлением в коде, используемом в конкретной СУБД, например в коде *ASCII*. Если сравниваются две строки символов, имеющие разные длины, более короткая строка дополняется справа пробелами для того, чтобы они имели одинаковую длину перед осуществлением сравнения.

Проверка на принадлежность значений заданному диапазону (оператор BETWEEN … AND). При этом проверяется, находится ли элемент данных между двумя заданными значениями.

Найти служащих, родившихся в интервале времени с 1 октября 1985 по 1 декабря 1990 года:

```
SELECT fname, lname FROM staff
WHERE dob BETWEEN to_date ('01.10.85', 'dd.mm.rr') AND
to_date('01.12.90', 'dd.mm.rr');
```

Следует отметить, что проверка на принадлежность диапазону не расширяет возможностей *SQL*, поскольку ее можно выразить в виде двух сравнений, т. е. выражение A BETWEEN B AND C эквивалентно (A>=B) AND (A<=C).

Помимо рассмотренной проверки в *SQL* существует также и инверсная проверка на принадлежность диапазону, реализованная с помощью оператора NOT BETWEEN … AND.

В стандарте *ANSI/ISO* определены относительно сложные правила обработки значений NULL в проверке BETWEEN … AND.

– Если проверяемое выражение имеет значение NULL либо оба выражения, определяющие диапазон, равны NULL, то проверка возвращает NULL.

– Если выражение, определяющее нижнюю границу диапазона, имеет значение NULL, то проверка возвращает FALSE, когда проверяемое значение больше верхней границы диапазона, и NULL в противном случае.

– Если выражение, определяющее верхнюю границу диапазона, имеет значение NULL, то проверка возвращает FALSE, когда проверяемое значение меньше нижней границы диапазона, и NULL в противном случае.

Проверка на членство во множестве (оператор IN). В этом случае проверяется, соответствует ли элемент данных какому-либо значению из заданного списка.

Следующий пример иллюстрирует процедуру проверки на членство во множестве. Вывести информацию об офисах, расположенных в Минске, Витебске и Бресте:

```
SELECT city, tel_no FROM branch
WHERE city IN ('Минск', 'Витебск', 'Брест');
```

Проверка IN также не добавляет новых возможностей, так как условие X IN (A, B, C) полностью эквивалентно условию (X = A) OR (X = B) OR (X = C).

Проверка на соответствие шаблону (оператор LIKE). Иногда для получения требуемых данных с выборкой по значениям строковых атрибутов в условии отбора инструкций на извлечение информации целесообразно применять шаблоны для поиска.

Следующий пример иллюстрирует применение проверки на соответствие шаблону. Вывести информацию обо всех сотрудниках, фамилии которых начинаются на букву К:

```
SELECT fname, address, tel_no FROM staff
WHERE fname LIKE 'К%';
```

Следует обратить внимание на запись шаблона строки сравнения в условии LIKE. Указанный способ задания строки шаблона характерен для диалекта *SQL* СУБД *Oracle* и отличен от регламентированного стандартом.

Символ процент «%» замещает произвольную последовательность символов, а символ подчеркивание «_» одиночный символ. Шаблоны, так же как и обыкновенные строчные данные, заключаются в одинарные кавычки. Более сложные шаблоны реализуются с помощью регулярных выражений (REGEXP_INSTR, REGEXP_LIKE, REGEXP_REPLACE, REGEXP_SUBSTR, REGEXP_COUNT).

Проверка на значение NULL (оператор IS [NOT] NULL). Иногда бывает необходимо проверять значения столбцов на содержание значения NULL и непосредственно их обрабатывать.

Например, найдем служащих, которые не закреплены ни за одним офисом:

SELECT fname FROM staff WHERE bno IS NULL;

Необходимо помнить, что NULL-значения не равны между собой в операциях сравнения, но при применении предиката DISTINCT считаются одинаковыми.

Обработка пустых значений. Если функции передается пустой аргумент, она возвращает NULL, за исключением следующих пяти случаев применения специальных встроенных функций:

1. Функция DECODE. Функция DECODE сравнивает свой первый аргумент с одним или несколькими поисковыми выражениями, которые спарены с результирующими выражениями:

SELECT first_name, DECODE (commission_pct, NULL, 'нет комиссионных', 0.4, commission_pct||' это максимум', 0.1, commission_pct||' это минимум', commission_pct) "КОМИССИОННЫЕ" FROM employees ORDER BY 2;

2. Функция NVL. Если ее первый аргумент есть NULL, функция NVL возвращает значение своего второго аргумента:

SELECT first_name, NVL (commission_pct, 0) "КОМИССИОННЫЕ"
FROM employees ORDER BY 2 DESC;

3. Функция NVL2. Функция расширяет функциональность NVL и позволяет заменяет значение, когда встречается NULL значение, а также когда встречается не-NULL значение.

4. Функция NULLIF. Сравнивает два своих обязательных аргумента и, если они равны, возвращает NULL, в противном случае, возвращает первый аргумент. Аргументы должны быть одного типа.

SELECT first_name, commission_pct, salary, NVL2 (NULLIF (commission_pct, salary*commission_pct), salary+salary*commission_pct||' зп+комиссия', salary||' зп') "КОМИССИОННЫЕ/ЗАРПЛАТА" FROM employees ORDER BY 2;

5. Функция COALESCE. Возвращает первое ненулевое выражение из списка. Если все выражения определены как NULL, то функция COALESCE вернет NULL.

SELECT first_name, COALESCE (commission_pct, 0) "КОМИССИОННЫЕ" FROM employees ORDER BY 2 DESC;

3.6. Многотабличные запросы

Одна из наиболее важных особенностей запросов *SQL* – это их способность определять связи между многочисленными таблицами и выводить всю информацию из них в терминах этих связей внутри одной команды. Этот вид операции называется соединением (JOIN).

Чтобы понять, как в *SQL* реализуются многотабличные запросы, рассмотрим запрос, который соединяет данные из двух различных таблиц. Составить список всех сотрудников, работающих в минском отделении.

```
SELECT fname, lname, position, s.tel_no FROM branch b, staff s
WHERE b.bno = s.bno AND city = 'Минск';
```

Представленный в примере синтаксис соединения является классическим синтаксисом соединения, который используется в СУБД *Oracle*. При отработке запроса СУБД должна произвести ряд действий, связанных с соединением таблиц *branch* и *staff*. Так, сначала просматривается столбец *city* с целью отбора строк со значениями, отличными от значения Минск, далее для отфильтрованных строк таблицы определяются значения столбца *bno* как идентификаторов минских отделений. После этого просматривается таблица *staff* с целью выявления строк со значениями в столбце *bno*, соответствующими идентификатору первого выявленного минского офиса. В найденных строках оставляются значения столбцов, указанных после ключевого слова SELECT. Далее эта же таблица просматривается для выявления строк, соответствующих второму минскому офису и т. д. до последнего. В итоге формируется таблица результатов с запрошенной информацией.

Отметим некоторые особенности многотабличных запросов. Как видно из примера, в многотабличном запросе часто используются полные имена столбцов, при этом в предложении FROM через пробел указываются псевдонимы таблиц, чтобы упростить обращение к столбцам по полному имени, а также обеспечить однозначность ссылок на столбцы. Кроме этого, особый смысл может иметь выбор всех столбцов (SELECT *), например в *Oracle* поддерживается следующий синтаксис:

```
SELECT s.*, city FROM staff s, branch b WHERE b.bno = s.bno;
```

Помимо соединения двух таблиц *SQL* допускает также соединение трех и более таблиц. Ограничений по количеству соединяемых таблиц ни стандарт, ни разработчики СУБД не предусматривают, однако следует иметь в виду, что при их увеличении в запросе снижается его «читабельность» и скорость выполнения в силу значительного увеличения затрат ресурсов и машинного времени при обработке.

В *OLTP*-приложениях запрос обычно ссылается только на одну или две таблицы. В этих приложениях время ответа является критичной величиной, пользователь, как правило, вводит один или два элемента данных, и ему требуется получить ответ от БД в течение одной или двух секунд.

В *OLAP*-приложениях, предназначенных для поддержки, принятия решений, запрос, как правило, обращается ко многим таблицам и использует сложные отношения, существующие в БД. В этих приложениях результаты запроса нужны для принятия важных решений, поэтому вполне приемлемыми считаются запросы, которые выполняются несколько минут и более.

Кроме того, комбинирование таблиц в запросе можно осуществлять с помощью операторов UNION – объединение, INTERSECT – пересечение, EXCEPT (MINUS) – разность.

При этом на таблицы накладываются определенные ограничения. Объединяемые таблицы должны быть совместимы, т. е. иметь одинаковое количество столбцов, а данные в столбцах должны быть одного типа и длины.

Обязанность проверять принадлежность данных к одному домену возлагается на пользователя. Общий синтаксис применения операторов комбинирования:

OPERATOR[ALL][CORRESPONDING[BY{*столбец* [,*столбец* …]}]];

Например, необходимо вывести список городов, в которых либо находится отделение компании, либо располагаются сдаваемые в аренду объекты:

```
SELECT city FROM branch WHERE city IS NOT NULL
UNION
SELECT city FROM property_for_rent WHERE city IS NOT NULL;
```

Отметим, что по умолчанию оператор UNION удаляет все дублирующиеся записи из разных наборов и получить полный набор записей можно только используя конструкцию UNION ALL.

Например, необходимо вывести список всех городов, в которых находятся и отделение компании, и сдаваемые в аренду объекты:

```
SELECT city FROM branch
INTERSECT
SELECT city FROM property_for_rent;
```

Например, нужно вывести список всех городов, в которых находятся отделения компании, но нет сдаваемых в аренду объектов:

```
SELECT city FROM branch
MINUS
SELECT city FROM property_for_rent;
```

Все эти примеры можно реализовать и по-другому, без использования операторов комбинирования запросов. Возможность использования нескольких эквивалентных форм запроса – один из самых существенных недостатков языка *SQL*.

3.7. Стандартный синтаксис внутренних и внешних соединений

Внутренние соединения. В стандарте *SQL* применяются отличные от рассмотренной выше синтаксические конструкции для соединения нескольких таблиц, которые могут быть использованы и в СУБД *Oracle*. Причем различают внешние и внутренние соединения, которые рассмотрим на примере таблиц

branch и *property_for_rent*, связанных отношением «один ко многим» по полю *bno*.

```
SELECT b. bno, b.city, pno, p.city
FROM branch b INNER JOIN property_for_rent p ON b.bno = p.bno;
```

Кроме этого, можно связать таблицы по другим полям с учетом их типов, размеров и логики сравнения. Например, можно записать инструкцию внутреннего соединения по полю *city* следующим образом:

```
SELECT b. bno, b.city, pno, p.city
FROM branch b INNER JOIN property_for_rent p ON b.city = p.city;
```

В этих случаях две таблицы соединяются явно посредством операции JOIN, а условие отбора, описывающее соединение, находится теперь в предложении ON внутри предложения FROM. В условии отбора могут быть заданы любые критерии сравнения строк двух соединяемых таблиц.

В таких простых двухтабличных соединениях все содержимое предложения WHERE просто переходит в предложение ON, т. е. последнее не добавляет ничего нового в язык *SQL*.

Стандарт *SQL* допускает еще один вариант запроса на простое внутреннее соединение таблиц. Так как связанные столбцы этих таблиц имеют одинаковые имена и сравниваются на предмет равенства (что делается довольно часто), то можно использовать альтернативную форму предложения ON, в которой задается список имен связанных столбцов:

```
SELECT *
FROM branch INNER JOIN property_for_rent USING (bno);
```

В предложении USING перечисляются через запятую имена связанных столбцов, которые должны быть идентичными в обеих таблицах. Использовать имя таблицы или синоним в USING запрещено. Это предложение полностью эквивалентно предложению ON, в котором каждая пара связанных столбцов задается явно, но намного компактнее и, следовательно, легче для понимания. Конечно, если связанные столбцы имеют разные имена в таблицах, то необходимо использовать предложение ON или WHERE со знаком равенства. Предложение ON требуется использовать также в том случае, если соединение не производится по равенству связанных столбцов. Например, если необходимо вывести информацию об отделениях и объектах недвижимости, находящихся в одних городах, но обсуживающихся в разных офисах, то необходимо задать соединение с помощью предложения ON:

```
SELECT b. bno, b.city, pno, p.city
FROM branch b INNER JOIN property_for_rent p ON b.bno <> p.bno AND
b.city = p.city;
```

Наконец, имеется еще один вариант этого простого запроса. Соединение двух таблиц, в котором связанные столбцы имеют идентичные имена, называется естественным соединением, так как обычно это действительно самый «естественный» способ соединения двух таблиц. Запрос на выборку отделений и объектов, которые в них обслуживаются и при этом находятся в одном городе и на одной улице, можно выразить как естественное соединение следующим образом:

```
SELECT * FROM branch NATURAL INNER JOIN property_for_rent;
```

Если задано ключевое слово NATURAL, предложения ON и USING могут отсутствовать в запросе на соединение, так как в естественном соединении точно определено условие отбора для соединения двух таблиц – сравниваются все столбцы с идентичными именами в обеих таблицах.

В стандарте *SQL* определено, что соединение двух таблиц по умолчанию является внутренним. Во всех предыдущих примерах можно опустить ключевое слово INNER, и полученные в результате запросы по-прежнему будут действительными.

Внешние соединения. Стандарт *SQL* обеспечивает полную поддержку внешних соединений, расширяя языковые конструкции, используемые для внутренних соединений. Например, полное внешнее соединение таблиц *Property_for_rent* и *Branch* создается следующим запросом:

```
SELECT b. bno, b.city, pno, p.city
FROM branch b FULL OUTER JOIN property_for_rent p ON b.bno = p.bno;
```

Таблица результатов запроса будет содержать по одной строке для каждой связанной по полю *bno* пары отделение/объект, а также по одной строке для каждой несвязанной равенством значений в поле *bno* записи для отделения или объекта, расширенной значениями NULL в столбцах другой таблицы. В стандарте *SQL* для внешних соединений допустимы те же самые вариации, что и для внутренних соединений, т. е. запрос можно было бы записать так:

```
SELECT *
FROM branch NATURAL FULL OUTER JOIN property_for_rent;
```

Или для поиска по одинаковому городу:

```
SELECT NVL (b.bno, 0) otdelenie, NVL (pno, 0) object
FROM branch b FULL OUTER JOIN property_for_rent p USING (city);
```

Ключевое слово OUTER также не является обязательным. По слову FULL СУБД сама определяет, что запрашивается внешнее соединение. В последнем примере для наглядности пустые значения в полях при выводе подменяются нулями и происходит операция соединения по полю *city*.

Левое и правое внешние соединения обозначаются словами LEFT и RIGHT вместо слова FULL. Вот вариант того же запроса, определяющий левое внешнее соединение:

```
SELECT NVL (b.bno, 0) otdelenie, NVL (pno, 0) object
FROM branch b LEFT JOIN property_for_rent p USING (city);
```

В таком запросе в результат войдут строки связанных пар отделение/объект и все несвязанные строки из таблицы *branch*, содержащие NULL для значений полей таблицы *property_for_rent*, но не войдут несвязанные строки из таблицы *property_for_rent*.

Правое внешнее соединение:

```
SELECT NVL (b.bno, 0) otdelenie, NVL (pno, 0) object
FROM branch b RIGHT JOIN property_for_rent p USING (city);
```

В данном случае в таблице результатов будут представлены пары отделение/объект и несвязанные строки из таблицы *property_for_rent* («правая» таблица соединения), содержащие значения NULL для несвязанных значений полей таблицы *branch*, но будут отсутствовать несвязанные строки из таблицы *branch*.

В традиционном синтаксисе *Oracle* для внешних соединений используется следующий синтаксис, например, для правого внешнего соединения:

```
SELECT s.*, city FROM staff s, branch b
WHERE b.bno = s.bno(+);
```

В случае левого соединения условие записывается, как WHERE b.bno(+) = = s.bno, а для получения полного внешнего соединения требуется объединить левое и правое соединение оператором UNION.

Расширенное предложение FROM поддерживает также два других способа соединения таблиц – декартово произведение и собственное соединение. Вот запрос, создающий декартово произведение таблиц *branch* и *property_for_rent*:

```
SELECT * FROM branch CROSS JOIN property_for_rent;
```

По определению, декартово произведение (называемое перекрестным соединением (CROSS JOIN)) содержит все возможные пары строк из двух таблиц. Оно является результатом «умножения» двух таблиц и превращает таблицы с данными, например, о трех отделениях и двух объектах в таблицу шести пар отделение/объект ($3 \cdot 2 = 6$). Перекрестным соединениям не сопутствуют никакие «связанные столбцы» или «условия отбора», поэтому предложения ON и USING в них не допускаются. Следует отметить, что операция перекрестного соединения не добавляет ничего нового к возможностям *SQL*. Те же результаты можно получить с помощью простого внутреннего соединения, если не задать в нем условия отбора. Поэтому предыдущий запрос можно переписать так:

```
SELECT * FROM branch, property_for_rent;
```

Ключевые слова CROSS JOIN в предложении FROM просто в явном виде указывают на то, что создается декартово произведение. Для большинства БД такая операция вряд ли будет представлять практический интерес. Она полезна

лишь тогда, когда на основе полученной таблицы строятся более сложные выражения, например итоговые запросы.

И, наконец, таблица может соединяться сама с собой – так называемое собственное соединение (*self-join*). Например, составим все возможные пары между сотрудниками:

```
SELECT s1.fname||'  и  '||s2.fname
FROM staff s1 JOIN staff s2 ON (s2.sno != s1.sno);
```

Основные выводы. Имена таблиц, из которых берутся данные, задаются в предложении FROM многотабличного запроса (соединения).

Каждая строка таблицы результатов запроса соединяет по одной строке из каждой исходной таблицы и является единственной строкой, содержащей комбинацию данных строк.

В многотабличных запросах чаще всего используются отношения предок/потомок, существующие между первичными и внешними ключами.

В общем случае соединения можно создавать путем сравнения любых пар столбцов из двух объединяемых таблиц, используя условие равенства или любое другое условие сравнения.

Таблицу можно соединять саму с собой, при этом необходимо использовать псевдоним таблицы.

Внешнее соединение является расширением стандартного (внутреннего) соединения, сохраняющим в таблице результатов запроса несвязанные строки одной или обеих исходных таблиц, при этом отсутствующие ячейки заполняются значениями NULL.

Стандарт *SQL* обеспечивает полную поддержку внутренних и внешних соединений и позволяет объединять результаты нескольких соединений при выполнении многотабличных операций.

На практике левое и правое внешние соединения более полезны, чем полное внешнее соединение, особенно если таблицы связаны через внешний и первичный ключи.

3.8. Предложения GROUP BY и HAVING

Запрос, включающий в себя предложение GROUP BY, называется запросом с группировкой, поскольку он объединяет строки исходных таблиц в группы и для каждой группы строк генерирует одну строку в таблице результатов запроса.

Столбцы, указанные в предложении GROUP BY, называются столбцами группировки, поскольку именно они определяют, по какому признаку строки делятся на группы. Приведем пример запроса с предложением GROUP BY. Пусть необходимо определить, сколько в среднем получают сотрудники в зависимости от занимаемой ими должности:

```
SELECT position, AVG(salary) AVG_salary FROM staff
GROUP BY position;
```

На логическом уровне запрос выполняется следующим образом.

1. Сотрудники делятся на группы, по одной группе для каждой занимаемой должности. В каждой группе все сотрудники имеют одно и то же значение в столбце *position*.

2. Для каждой группы вычисляется среднее значение столбца *salary* по всем строкам, входящим в группу, и генерируется одна итоговая строка результатов. Эта строка содержит значение столбца *position* для группы и среднюю зарплату сотрудников для данной группы.

Следует также отметить, что *SQL* допускает употребление в итоговых запросах нескольких столбцов группировки.

На запросы, в которых используется группировка, накладываются дополнительные ограничения:

– столбцы с группировкой должны представлять собой реальные столбцы таблиц, перечисленные в предложении FROM;

– нельзя группировать строки на основе вычисляемого выражения или псевдонима столбца.

Кроме того, существуют ограничения на элементы списка возвращаемых столбцов. Все элементы этого списка должны иметь одно значение для каждой группы строк. Это означает, что возвращаемым столбцом может быть:

– константа;

– статистическая функция, возвращающая одно значение для всех строк, входящих в группу;

– столбец группировки, который по определению имеет одно и то же значение во всех строках группы;

– выражение, включающее в себя перечисленные выше элементы.

На практике в список возвращаемых столбцов запроса с группировкой всегда входят столбец группировки и статистическая функция. Если последняя не указана, следовательно, запрос можно более просто выразить с помощью предиката DISTINCT без использования предложения GROUP BY. И наоборот, если не включить в результаты запроса столбец группировки, то невозможно будет определить, к какой группе относится каждая строка результатов. Другими словами, все имена столбцов в SELECT должны присутствовать и во фразе GROUP BY, за исключением случаев, когда имя столбца используется в обобщающей функции. Обратное утверждение неверно.

В стандарте *ANSI*/*ISO* определено, что два значения NULL в предложении GROUP BY равны. Если две строки имеют значение NULL в одинаковых столбцах группировки и идентичные значения во всех остальных столбцах группировки, то они помещаются в одну группу.

Условие отбора групп (предложение HAVING). Точно так же, как предложение WHERE используется для отбора отдельных строк, участвующих в запросе, предложение HAVING можно применить для отбора групп строк. Его формат соответствует формату предложения WHERE, т. е. состоит из ключевого слова HAVING, за которым следует условие отбора. Рассмотрим пример. Пусть

необходимо подсчитать количество сотрудников, работающих в каждом из офисов, исключив офисы, в которых работает менее двух человек:

```
SELECT bno, count (sno) Kolichestvo FROM staff
GROUP BY bno
HAVING COUNT (sno) >= 2;
```

Отметим, что в предложениях GROUP BY и HAVING не используются псевдонимы.

Ограничения на условия отбора групп. Предложение HAVING используется для того, чтобы включать и исключать группы строк из результатов запроса, поэтому используемое в нем условие отбора применяется не к отдельным строкам, а к группе в целом, т. е. используется только совместно с GROUP BY. Это значит, что в условие отбора может входить:

– константа;

– статистическая функция, возвращающая одно значение для всех строк, входящих в группу;

– столбец группировки, который по определению имеет одно и то же значение во всех строках группы;

– выражение, включающее в себя перечисленные выше элементы.

На практике условие отбора предложения HAVING всегда должно включать в себя как минимум одну статистическую функцию. Если это не так, следовательно, условие отбора можно переместить в предложение WHERE. Чтобы определить, где следует указать условие отбора – в предложении WHERE или HAVING – необходимо вспомнить, как применяются эти предложения:

– предложение WHERE применяется к отдельным строкам, поэтому выражения, содержащиеся в нем, должны вычисляться для отдельных строк и в нем никогда не применяются статистические функции;

– предложение HAVING применяется к группам строк, поэтому выражения, содержащиеся в нем, должны вычисляться для групп строк. При этом стандарт требует, чтобы имена столбцов, использующиеся в HAVING, обязательно присутствовали в списке GROUP BY или применялись в статистических функциях.

Как и условия отбора в предложении WHERE, условие отбора в предложении HAVING может дать один из следующих результатов:

– если условие отбора имеет значение TRUE, группа строк остается и для нее генерируется одна строка результатов запроса;

– если условие отбора имеет значение FALSE или NULL, группа строк исключается и строка для нее в результатах запроса не генерируется.

3.9. Статистические функции

Для подведения итогов по информации, содержащейся в БД, в *SQL* предусмотрены статистические (агрегатные) функции. Статистическая функция принимает в качестве аргумента какой-либо столбец данных целиком, а возвращает одно значение, которое определенным образом подытоживает этот столбец, исключая все пустые значения.

В стандарте *SQL* указано пять статистических функций, которые позволяют получать различные виды итоговой информации:

– SUM (*имя_столбца*) – вычисляет суммы всех значение столбца-аргумента;

– AVG (*имя_столбца*) – вычисляет среднее значения столбца;

– MIN (*имя_столбца*) – определяет минимальное значение столбца;

– MAX (*имя_столбца*) – определяет максимальное значение столбца;

– COUNT (*имя_столбца*) – подсчитывает число всех определенных значений столбца, в варианте COUNT (*) – подсчитывает число всех строк таблицы.

Аргументом статистической функции может быть простое имя столбца или выражение.

В *Oracle* применяется более 25 статистических функций, которые можно разделить на агрегатные и аналитические (оконные) функции. Все агрегатные функции возвращают только одно значение для каждого набора строк, а аналитические могут возвращать несколько строк для каждой группы.

Аналитические функции агрегируют данные порциями (*partitions*), количество и размер которых можно регулировать специальной синтаксической конструкцией OVER ([PARTITION …] ORDER BY *выражение* 2 [,…] [{ASC/DESC}] [{NULLS FIRST/NULLS LAST}]).

Множество оконных функций можно разделить на три класса:

– агрегирующие (*aggregate*) – это может быть любая агрегатная функция, рассмотренная выше;

– ранжирующие (*ranking*) – это функции, которые позволяют установить ранги строк в зависимости от имеющихся в них значений, например ROW_NUMBER (), RANK (), DENSE_RANK (), PERCENT_RANK (). Обязательным для них является указание условия ORDER BY, по которому будет происходить сортировка ранжирования, а значения NULL всегда будут определяться одинаковым рангом.

– функции смещения (*value*) – это функции, которые позволяют, перемещаясь по выделенной партиции таблицы, обращаться к предыдущему значению строки или крайним значениям строк в партиции, например LAG (), LEAD (), FIRST_VALUE (), LAST_VALUE ().

Более подробно синтаксис и применение аналитических функций можно самостоятельно изучить, используя источники [5], [10].

Ниже рассмотрены примеры запросов только со стандартными агрегатными функциями. Пусть необходимо определить среднемесячную и среднегодовую зарплату сотрудников предприятия:

```
SELECT AVG (salary) AS per_month, AVG (12*salary) AS per_year
FROM staff;
```

В примере в списке возвращаемых столбцов результирующего множества используются псевдонимы столбцов, иначе значения в возвращаемой строке были бы представлены под именами AVG (salary) и AVG (12*salary).

Тип данных столбца-аргумента функции COUNT () может быть любым (не обязательно числовым, в отличие от других статистических функций), а возвращает она всегда целое число независимо от типа данных столбца, т. е. никогда не сможет вернуть NULL: при отсутствии строк вернет 0. Например, следующий запрос подсчитывает количество работающих на предприятии менеджеров:

```
SELECT COUNT (sno) manager
FROM staff
WHERE INITCAP (position)='Manager';
```

Следует заметить, что для функции COUNT () важны не конкретные значения ячеек, а количество ячеек, удовлетворяющих заданному критерию.

В отличие от функции COUNT () статистическая функция COUNT (*) подсчитывает строки, а не значения в них. Функция COUNT () не учитывает NULL-значения в столбцах, так же как и любая другая статистическая функция.

Кроме того, здесь используется встроенная символьная функция INITCAP, чтобы гарантировать соответствие данных в поле заданному шаблона поиска.

В *SQL* существует ряд ограничений на применение статистических функций в инструкциях SELECT:

1) в списке возвращаемых столбцов можно указывать любую статистическую функцию, она также может входить в выражение, в котором суммируются или вычитаются значения двух и более статистических функций, однако статистическая функция не может быть аргументом для другой такой функции, поскольку получающееся в данном случае выражение не имеет смысла;

2) в списке возвращаемых столбцов нельзя одновременно указывать статистическую функцию и обычные имена столбцов, поскольку в этом также нет смысла;

3) разрешается использовать предикат DISTINCT во всех статистических функциях, но в одном запросе его можно употребить только один раз. Если он применяется вместе с аргументом одной из статистических функций, его нельзя использовать ни с одним другим аргументом. Если он указан перед списком возвращаемых столбцов, его нельзя употреблять ни в одной статистической функции. Единственным исключением из этого правила является случай, когда предикат DISTINCT используется внутри подчиненного запроса, входящего в запрос, в котором он уже применяется.

Таким образом, статистические функции в основном могут использоваться только в предложениях SELECT и HAVING, при чем, если они есть в SELECT, а в GROUP BY нет, то никакие другие столбцы в списке SELECT быть не могут.

В стандарте *ANSI*/*ISO* сказано, что значения NULL статистическими функциями игнорируются и определены следующие точные правила обработки значений NULL в статистических функциях:

– если какие-либо из значений, содержащихся в столбце, равны NULL, при вычислении результата функции они исключаются;

– если все значения в столбце равны NULL, то функции SUM (), AVG (), MIN () и MAX () возвращают значение NULL; функция COUNT () возвращает ноль;

– если в столбце нет значений (т. е. столбец пустой), то функции SUM (), AVG (), MIN () и MAX () возвращают значение NULL; функция COUNT () возвращает ноль;

– функция COUNT (*) подсчитывает количество строк и не зависит от наличия или отсутствия в столбце значений NULL; если строк в таблице нет, эта функция возвращает ноль.

Хотя эта часть стандарта определена достаточно четко, коммерческие СУБД могут выдавать результаты, отличающиеся от стандарта, особенно если все значения, содержащиеся в столбце, равны NULL или таблица пуста. Поэтому, прежде чем полагаться на стандартные правила, следует проверить используемую СУБД.

3.10. Подчиненные запросы (подзапросы)

Подчиненным называется запрос, содержащийся в предложении WHERE или HAVING другой инструкции *SQL*. В предложении WHERE они помогают отбирать из таблицы результатов запроса отдельные строки, а в предложении HAVING – группы строк. Кроме этого, подчиненные запросы используются в инструкциях INSERT, UPDATE, DELETE, а также в списке столбцов инструкции SELECT или во FROM. В разделе FROM может быть бесконечное количество вложенных подзапросов, но в WHERE – максимум 255.

Механизм подчиненных запросов позволяет использовать результаты одного запроса в качестве составной части другого.

По типу возвращаемых значений существует три типа подзапросов:

1) скалярный – возвращает единственное значение;

2) строковый – возвращает значения нескольких столбцов таблицы, но в одной строке;

3) табличный – возвращает значения одного и более столбцов таблицы, размещенные более чем в одной строке.

В связи с этим в терминологии *Oracle* принято выделять три вида подзапросов: однострочные (SINGLE_ROW), многострочные (MULTIPLE-ROW), коррелированные (CORRELATED).

Понятие подчиненного запроса играет важную роль в *SQL* по трем следующим причинам:

– инструкция *SQL* с подчиненным запросом часто является самым естественным способом выражения запроса, так как она лучше всего соответствует его словесному описанию;

– подчиненные запросы облегчают написание инструкции SELECT, поскольку они позволяют разбивать запрос на части (на запрос и подчиненные запросы);

– существуют запросы, которые нельзя сформулировать на *SQL* иначе как с помощью подзапросов.

Подчиненный запрос всегда заключается в круглые скобки и сохраняет структуру инструкции SELECT, содержащей предложение FROM и необязательные предложения WHERE, GROUP BY и HAVING. Структура этих предложений в подчиненном запросе идентична их структуре в инструкции SELECT; в подчиненном запросе эти предложения выполняют свои обычные функции.

Однако между подчиненным запросом и инструкцией SELECT в соответствии со стандартом имеется ряд отличий:

– Таблица результатов подчиненного запроса должна состоять только из одного столбца. Это означает, что в предложении SELECT подчиненного запроса должен указываться только один возвращаемый столбец (*Oracle* отходит от стандарта и использует в результирующем наборе более одного столбца).

– В подчиненный запрос не может входить предложение ORDER BY. Результаты подчиненного запроса используются только внутри главного запроса и для пользователя остаются невидимыми, поэтому нет смысла их сортировать.

– Имена столбцов в подчиненном запросе могут являться ссылками (внешними ссылками) на столбцы таблиц главного запроса.

– Подчиненный запрос не может быть запросом на объединение нескольких различных инструкций SELECT (в *Oracle* может).

– Подчиненный запрос в операции сравнения может стоять только справа от оператора сравнения, т. е. неравенство A < (подчиненный запрос) разрешается, а неравенство (подчиненный запрос) > A недопустимо. Сказанное не ограничивает возможности операции сравнения, поскольку знак любого неравенства всегда можно «перевернуть» так, чтобы подчиненный запрос оказался с правой стороны. Однако это говорит о том, что иногда требуется «переворачивать» логику словесного запроса так, чтобы он формально соответствовал действительной инструкции стандартного *SQL* (в *Oracle* допускается размещение с любой стороны).

Чаще всего подчиненные запросы указываются в предложении WHERE. Когда подчиненный запрос содержится в данном предложении, он участвует в процессе отбора строк. В простейшем случае подчиненный запрос является частью условия отбора и возвращает значение, позволяющее проверить истинность или ложность условия.

Условия отбора в подчиненном запросе. В *SQL* используются следующие условия отбора в подчиненном запросе:

– сравнение с результатом подчиненного запроса. Значение выражения сравнивается с одним значением, которое возвращается подчиненным запросом;

– проверка на принадлежность результатам подчиненного запроса. Значение выражения проверяется на равенство одному из множества значений, которые возвращаются подчиненным запросом. Эта проверка напоминает простую проверку на членство во множестве;

– проверка на существование. Проверяется наличие строк в таблице результатов подчиненного запроса;

– многократное сравнение. Значение выражения сравнивается с каждым из множества значений, которые возвращаются подчиненным запросом.

Сравнение с результатом подчиненного запроса (операторы =, !=, <, <=, >, >=). Сравнение с результатом подчиненного запроса является модифицированной формой простого сравнения. Значение выражения сравнивается со значением, которое возвращается подчиненным запросом, и если условие сравнения выполняется, то проверка дает результат TRUE. Эта проверка используется для сравнения значения из проверяемой строки с одним значением, полученным от подчиненного запроса, как показано в примере. Пусть необходимо вывести список сотрудников, заработная плата которых превышает среднюю заработную плату сотрудников всей компании, и определить, сколько составляет это превышение:

```
SELECT sno, fname, lname, position, salary - (SELECT AVG (salary)
FROM staff) AS sal_diff FROM staff
WHERE salary > (SELECT AVG (salary) FROM staff);
```

Подчиненный запрос, участвующий в операции сравнения, должен возвращать в качестве результата единичное значение, т. е. одну строку, содержащую один столбец. Если результатом подчиненного запроса являются несколько строк или несколько столбцов, то сравнение не имеет смысла и СУБД выдает сообщение об ошибке. Если в результате выполнения подчиненного запроса не будет получено ни одной строки или будет получено значение NULL, то операция сравнения возвращает NULL.

Проверка на принадлежность результатам подчиненного запроса (предикат IN). Эта проверка является видоизмененной формой простой проверки на членство во множестве. Одно значение сравнивается со столбцом данных, которые возвращаются подчиненным запросом, и, если это значение равно одному из элементов столбца, проверка дает результат TRUE. Данная проверка используется, когда необходимо сравнить значение из проверяемой строки с множеством значений, отобранных подчиненным запросом. Рассмотрим пример, где необходимо вывести список сотрудников, работающих в минских отделениях компании:

```
SELECT fname, lname, position FROM staff
WHERE bno IN (SELECT bno FROM branch
                    WHERE city = 'Минск');
```

Этот подчиненный запрос возвращает список идентификаторов отделений, расположенных в Минске. Главный (внешний запрос) посредством проверки IN возвращает выборочную информацию о сотрудниках из тех строк таблицы *staff*, которые по значению в *bno* удовлетворяют ее истинности.

Следует отметить, что в запросах с подчинением возможно употребление также и инверсной формы проверки на принадлежность результатам подчиненного запроса посредством предиката NOT IN.

Проверка на существование (предикат EXISTS). В результате проверки на существование (предикат EXISTS) можно выяснить, содержится ли в таблице результатов подчиненного запроса хотя бы одна строка. Аналогичной простой проверки не существует. Проверка на существование допустима только в подчиненных запросах.

Если подчиненный запрос не возвращает ни одной строки, проверка EXISTS возвращает значение FALSE. Эта проверка не может возвращать NULL. Можно изменить логику проверки EXISTS и использовать форму NOT EXISTS.

Предикат EXISTS в действительности совсем не использует результаты подчиненного запроса. Проверяется только наличие результатов. По этой причине в *SQL* смягчается правило, согласно которому «подчиненный запрос должен возвращать один столбец данных», и в подчиненном запросе проверки EXISTS допускается использование формы SELECT *, что иногда наблюдается на практике, однако чаще всего для экономии указывается просто 1, как в примере.

Рассмотрим еще один вариант создания предыдущего вопроса, например выведем список сотрудников, работающих в минских отделениях компании:

```
SELECT sno, fname, lname, position FROM staff s
WHERE EXISTS (SELECT 1 FROM branch b
                    WHERE s.bno = b.bno AND city = 'Минск');
```

Отметим, что в приведенном примере подчиненный запрос содержит внешнюю ссылку на столбец таблицы из главного запроса (*s.bno*). Внешняя ссылка представляет собой имя столбца, не входящего ни в одну из таблиц, перечисленных в предложении FROM подчиненного запроса, и принадлежащего таблице, указанной в предложении FROM главного запроса. На практике в подчиненном запросе проверки EXISTS всегда имеется внешняя ссылка, «связывающая» подчиненный запрос со строкой, проверяемой в настоящий момент главным запросом. Запросы с внешней ссылкой также называют коррелированными, и этот прием может быть использован в других видах подзапросов (IN, ANY/ALL).

Многократное сравнение (предикаты ANY и ALL). В *SQL* имеются также две разновидности многократного сравнения – ANY и ALL, расширяющие

предыдущие проверки до уровня других операторов сравнения, таких как больше (>) или меньше (<). В обеих проверках некоторое значение сравнивается со столбцом данных, отобранных подчиненным запросом.

Предикат ANY. В проверке ANY, для того чтобы сравнить проверяемое значение со столбцом данных, отобранных подчиненным запросом, используется один из шести операторов сравнения (=, !=, <, <=, >, >=). Проверяемое значение поочередно сравнивается с каждым элементом, содержащимся в столбце. Если любое из этих сравнений дает результат TRUE, то проверка ANY возвращает значение TRUE. Для демонстрации запроса с предикатом ANY определим имя и номер телефона потенциальных арендаторов, чьи требования по арендной стоимости соответствуют предварительной арендной стоимости хотя бы одной из предлагаемых в аренду двухкомнатных квартир:

```
SELECT fname, lname, tel_no FROM renter
WHERE max_rent >= ANY (SELECT rent FROM property_for_rent
                       WHERE type = 'f' AND rooms = 2);
```

В соответствии со стандартом *ANSI*/*ISO* вместо предиката ANY можно использовать предикат SOME. Обычно можно употреблять любой из них, но некоторые СУБД не поддерживают предикат SOME.

Если подчиненный запрос в проверке ANY не создает ни одной строки или если столбец результатов содержит значения NULL, то в различных СУБД проверка ANY может выполниться по-разному. В стандарте *ANSI*/*ISO* для языка *SQL* содержатся подробные правила, определяющие результаты проверки ANY, когда проверяемое значение сравнивается со столбцом результатов подчиненного запроса.

Если подчиненный запрос возвращает результат в виде пустого столбца, то проверка ANY возвращает значение FALSE (в результате выполнения подчиненного запроса не получено ни одного значения, для которого выполнялось бы условие сравнения).

Если операция сравнения имеет значение TRUE хотя бы для одного значения в столбце, то проверка ANY возвращает значение TRUE (имеется некоторое значение, полученное подчиненным запросом, для которого условие сравнения выполняется).

Если операция сравнения имеет значение FALSE для всех значений в столбце, то проверка ANY возвращает значение FALSE (можно утверждать, что ни для одного значения, возвращенного подчиненным запросом, условие сравнения не выполняется).

Если операция сравнения не имеет значение TRUE ни для одного значения в столбце и в нем имеется одно или несколько значений NULL, то проверка ANY возвращает результат NULL. (В этой ситуации невозможно с определенностью утверждать, существует ли полученное подчиненным запросом значение, для которого выполняется условие сравнения; может быть, существует, а может, и нет – все зависит от «настоящих» значений неизвестных данных.)

На практике проверка ANY иногда может приводить к ошибкам, которые трудно выявить, особенно когда применяется оператор сравнения «не равно» (!=).

Запрос с предикатом ANY всегда можно преобразовать в запрос с предикатом EXISTS, перенося операцию сравнения внутрь условия отбора подчиненного запроса.

Предикат ALL. В проверке ALL, как и в проверке ANY, используется один из шести операторов сравнения проверяемого значения со столбцом данных, отобранных подчиненным запросом. Проверяемое значение поочередно сравнивается с каждым элементом, содержащимся в столбце. Если все сравнения дают результат TRUE, то проверка ALL возвращает значение TRUE. Для демонстрации запроса с предикатом ALL пусть необходимо определить имя и номер телефона потенциальных арендаторов, чьи требования по арендной стоимости соответствуют предварительной арендной стоимости любого из предлагаемых в аренду объектов недвижимости типа двухкомнатная квартира:

```
SELECT fname, lname, tel_no FROM renter
WHERE max_rent >= ALL (SELECT rent  FROM property_for_rent
                       WHERE type = 'f' AND rooms = 4);
```

Проверка ALL, подобно проверке ANY, может оказаться трудной для понимания, поскольку включает в себя не одно сравнение, а несколько.

Если подчиненный запрос в проверке ALL не возвращает ни одной строки или если столбец результатов запроса содержит значения NULL, то в различных СУБД проверка ALL может выполняться по-разному. В стандарте *ANSI*/*ISO* для языка *SQL* содержатся подробные правила, определяющие результаты проверки ALL, когда проверяемое значение сравнивается со столбцом результатов подчиненного запроса. Если подчиненный запрос возвращает результат в виде пустого столбца, то проверка ALL возвращает значение TRUE. Считается, что условие сравнения выполняется, даже если результаты подчиненного запроса отсутствуют.

Если операция сравнения дает результат FALSE для какого-нибудь значения в столбце, то проверка ALL возвращает значение FALSE. В этом случае можно утверждать, что условие сравнения выполняется не для каждого значения, возвращенного подчиненным запросом.

Если операция сравнения не дает результат FALSE ни для одного значения в столбце, но для одного или нескольких значений дает результат NULL, то проверка ALL возвращает значение NULL. В этой ситуации нельзя с определенностью утверждать, для всех ли значений, возвращенных подчиненным запросом, справедливо условие сравнения; может быть, для всех, а может, и нет – все зависит от «настоящих» значений неизвестных данных.

Ошибки, которые могут случиться, если проверка ANY содержит оператор сравнения «не равно» (< >), происходят и в проверке ALL. Проверку ALL, так же

как и проверку ANY, всегда можно преобразовать в эквивалентную проверку на существование (EXISTS), перенеся операцию сравнения в подчиненный запрос.

Уровни вложенности запросов. Все рассмотренные до сих пор запросы были «двухуровневыми» и состояли из главного и подчиненного запросов. Точно так же, как внутри главного запроса может находиться подчиненный запрос, внутри подчиненного запроса может находиться еще один подчиненный запрос, называемый в таком случае вложенным. Вот пример «трехуровневого» запроса: составить перечень сдаваемых в аренду объектов, за которые отвечают работники отделения компании, расположенного на улице, начинающейся на букву K:

```
SELECT street, city, type, rooms, rent FROM property_for_rent
WHERE sno IN (SELECT sno FROM staff
                    WHERE bno IN (SELECT bno  FROM branch
                                        WHERE street LIKE '%K%'));
```

По такой же методике можно создавать запросы с четырьмя и более уровнями вложенности. Стандарт *ANSI*/*ISO* не определяет максимальное число уровней вложенности, но на практике с ростом их числа очень быстро увеличивается время выполнения запроса. Когда запрос имеет более двух уровней вложенности, он становится трудным для чтения и понимания. Во многих СУБД количество уровней вложенности запросов ограничено относительно небольшим числом.

3.11. Добавление данных (инструкция INSERT INTO)

Наименьшей единицей информации, которую можно добавить в реляционную БД, является одна строка. Рассмотрим три способа добавления новых строк в БД.

– Однострочная инструкция INSERT позволяет добавить в таблицу одну новую строку. Она широко используется в повседневных приложениях, например в программах ввода данных.

– Многострочная инструкция INSERT обеспечивает извлечение строк из одной части БД и добавление их в другую таблицу. Она обычно используется, когда «старые» строки таблицы пересылаются в неиспользуемую таблицу для сохранения.

– Инструкция MERGE.

Однострочная инструкция INSERT INTO. Однострочная инструкция INSERT INTO добавляет в таблицу новую строку.

```
INSERT INTO имя_таблицы [(список_столбцов)]
VALUES (список_значений_данных);
```

В предложении INTO указывается имя таблицы или представления, в которые добавляется новая строка, а в предложении VALUES содержатся значения

данных для новой строки. Список столбцов определяет, какие значения в какой столбец заносятся.

На практике данные почти всегда добавляются в базу с помощью программ ввода данных, использующих специальные формы. После окончания ввода данных в форму приложение добавляет новую строку данных в базу с помощью программного *SQL*. Однако инструкцию INSERT можно использовать и в интерактивном режиме. Независимо от того, какой *SQL* используется, интерактивный или программный, инструкция INSERT имеет один и тот же вид.

Как правило, в инструкции INSERT указывается простое имя таблицы, определяющее вашу собственную таблицу. Чтобы вставить данные в таблицу, созданную другим пользователем, необходимо указать ее полное имя, т. е. указать ее имя вместе с именем схемы. Также необходимо иметь разрешение на ввод данных в эту таблицу, иначе выполнение инструкции закончится неудачей.

Список столбцов в инструкции INSERT – необязательный параметр, он служит для того, чтобы установить соответствие между значениями данных, содержащимися в предложении VALUES, и столбцами, для которых эти данные предназначены. Списки значений и столбцов должны содержать одинаковое число элементов, а тип данных каждого значения должен быть совместимым с типом соответствующего столбца. Если списка столбцов нет, то используются все столбцы в том порядке, в котором они перечислены в CREATE TABLE.

Добавление значений NULL. При добавлении в таблицу новой строки всем столбцам, имена которых отсутствуют в списке столбцов инструкции INSERT, автоматически присваивается значение NULL.

Значение NULL можно присвоить и явным образом, включив эти столбцы в список, а в списке значений – задав для них ключевое слово NULL, например допустим, что необязательным является указание должности:

```
INSERT INTO staff (sno, fname, lname, address, tel_no, position, sex, salary, dob, bno)
VALUES (6, 'Иванов', 'Иван', 'Кирова 34', '6455811', NULL, 'm', 5000, TO_DATE
('05.02.92', 'dd.mm.rr'), 2);
```

Добавление всех столбцов. Для удобства в *SQL* разрешается не включать список столбцов в инструкцию INSERT. Если список столбцов опущен, он генерируется автоматически и в нем слева направо перечисляются все столбцы таблицы. При выполнении инструкции SELECT * генерируется такой же список столбцов. Пользуясь этой сокращенной формой записи, предыдущую инструкцию можно переписать таким образом:

```
INSERT INTO staff
VALUES (6, 'Иванов', 'Иван', ' Кирова 34', '6455811', NULL, 'M', TO_DATE
('05.02.92', 'dd.mm.rr'), 5000, 2);
```

Как видно, если список столбцов опущен, то в списке значений явно указаны значения NULL, а последовательность значений данных в точности соответствует порядку столбцов в таблице.

Многострочная инструкция INSERT INTO. Многострочная инструкция INSERT INTO добавляет в таблицу несколько строк путем копирования множества строк одной таблицы в другую таблицу.

INSERT INTO *имя_таблицы* [(*список_столбцов*)]
SELECT *запрос*;

В этой разновидности инструкции INSERT значения для новых строк явно не задаются. Источником новых строк служит запрос на выборку (SELECT), содержащийся внутри инструкции.

На запрос, содержащийся внутри многострочной инструкции INSERT, накладывается несколько ограничений:

– в запрос нельзя включать предложение ORDER BY. Не имеет смысла сортировать таблицу результатов запроса, поскольку она добавляется в таблицу, которая, как и все остальные, не упорядочена;

– таблица результатов запроса должна содержать количество столбцов, равное длине списка столбцов в инструкции INSERT (или полностью повторять список столбцов таблицы, если список столбцов отсутствует), а типы данных соответствующих столбцов таблицы результатов запроса и целевой таблицы должны быть совместимыми;

– имя таблицы, указанное в инструкции INSERT, не может присутствовать в предложении FROM запроса на выборку или любого запроса, вложенного в него. Тем самым запрещается добавление таблицы самой к себе.

Например, пусть существует таблица *staff_prop_count* (*sno*, *fname*, *lname*, *prop_count*), которая содержит имена работников компании и их учетные номера, а также количество объектов аренды, за которые они отвечают.

```
INSERT INTO staff_prop_count
SELECT s.sno, fname, lname, COUNT (p.sno)
FROM staff s, property_for_rent p
WHERE s.sno = p.sno
GROUP BY s.sno, fname, lname
UNION
SELECT s.sno, fname, lname,0
FROM staff s
WHERE s.sno NOT IN (SELECT DISTINCT p.sno
FROM property_for_rent p);
```

Отметим, что инструкция UNION в таком применении поддерживается не во всех диалектах *SQL*.

Если необходимо вставить в таблицу в интерактивном режиме *SQL* несколько новых строк одновременно, то можно воспользоваться инструкцией INSERT ALL.

INSERT ALL
INTO *имя_таблицы* [(*список_столбцов*)] VALUES (*значение* [, *значение*] ...)
INTO *имя_таблицы* [(*список_столбцов*)] VALUES (*значение* [, *значение*] ...)
…
SELECT * FROM *dual*;

Инструкция MERGE. Начиная с версии *Oracle* 9*i*, появилась возможность использовать инструкцию MERGE INTO, которая объединяет в себе возможности инструкций вставки, обновления и удаления, делая их более удобными для пользователей.

MERGE INTO – это *DML*-оператор вставки (INSERT), обновления (UPDATE) и удаления (DELETE, начиная с *Oracle Database* 10*g*) данных при слиянии таблиц. Когда выполняется оператор MERGE, в сервере *Oracle* 9*i Database* немедленно срабатывают триггеры BEFORE UPDATE и BEFORE INSERT, так как в этом операторе есть предложения WHEN MATCHED THEN UPDATE (когда совпадают, то обновить) и WHEN NOT MATCHED THEN INSERT (когда не совпадают, то вставить), которые являются обязательными. В сервере *Oracle Database* 10*g* и выше будут срабатывать триггеры BEFORE UPDATE, INSERT и/или DELETE – в зависимости от указанных в операторе MERGE операций.

Общий синтаксис инструкции MERGE INTO:

MERGE INTO *таблица_назначения приемник*
USING (SELECT *столбец1*, *столбец2*, *столбец3*
FROM *таблица_источник*) *источник*
ON (*приемник. столбец1 = источник. столбец1*)
WHEN MATCHED THEN
UPDATE SET *приемник.столбец2 = источник.столбец2*,
приемник.столбец3 = источник.столбец3
WHERE *источник.столбец2* IS NOT NULL
DELETE *источник.столбец*2 IS NULL
WHEN NOT MATCHED THEN
INSERT (*приемник.столбец1*, *приемник.столбец*2, *приемник.столбец*3)
VALUES (*источник.столбец1*, *источник.столбец2*, *источник.столбец3*)
WHERE *источник.столбец2* IS NOT NULL;

По сути, это простое слияние таблиц *таблица_назначения* и *таблица_источник*, для которых проверяется условие *приемник.столбец1 = источник.столбец1*, и если оно истинно, то выполняется UPDATE, если нет – INSERT. Причем нельзя изменять поля таблицы *таблица_назначения* в секции UPDATE, по которым идет связывание двух таблиц. В качестве *таблица_источник* чаще всего используется подзапрос.

Например, в дополнительной таблице *copy_staff* будем обновлять фамилию, имя и зарплату сотрудников, если они уже есть в этой таблице, иначе – добавлять новых сотрудников:

```
MERGE INTO copy_staff d
USING (SELECT * FROM staff) s
ON (d.sno = s.sno)
WHEN MATCHED THEN
UPDATE SET
d.fname = s.fname,
d.lname = s.lname,
d.salary = s.salary
WHEN NOT MATCHED THEN
INSERT VALUES (s.sno, s.fname, s.lname, s.adress, s.tel_no, s.position, s.sex, s.dob, s.salary, s.bno);
```

Назначение операции MERGE – ускорить обновление больших таблиц. Обратите внимание, что обновление выполняется применительно к одному состоянию БД и поэтому не сводимо к последовательному выполнению команд INSERT, UPDATE и DELETE.

3.12. Удаление данных (инструкция DELETE)

Инструкция DELETE удаляет выбранные строки из одной таблицы.

```
DELETE FROM имя_таблицы
[WHERE условие];
```

В предложении FROM указывается таблица или представление, содержащие строки, которые требуется удалить. В предложении WHERE указывается критерий отбора строк, которые должны быть удалены.

Предположим, что недавно принятый на работу Иванов решил уволиться из компании. Инструкция DELETE, удаляющая относящуюся к служащему строку из таблицы *staff*, выглядит следующим образом:

```
DELETE FROM staff
WHERE lname = 'Иванов';
```

Условия отбора, которые можно задать в предложении WHERE инструкции DELETE, полностью совпадают с условиями отбора, доступными в одноименном предложении инструкции SELECT, рассмотренными выше.

Удаление всех строк. Хотя предложение WHERE в инструкции DELETE является необязательным, оно присутствует почти всегда. Если же оно отсутствует, то удаляются все таблицы. Например, удалить информацию обо всех сотрудниках:

```
DELETE FROM staff;
```

В результате выполнения приведенной инструкции таблица *staff* становится пустой, но из БД она не удаляется, так как определение таблицы и ее столбцов остается и в нее по-прежнему можно добавлять новые строки. Чтобы удалить из БД определение таблицы, необходимо использовать *DDL*-инструкцию DROP TABLE.

Инструкция DELETE, приведенная в примере, несет в себе потенциальную угрозу удаления необходимых строк, поэтому всегда следует задавать условие отбора и обращать внимание на то, отбирает ли оно действительно ненужные строки. Желательно вначале проверить предложение WHERE в интерактивном режиме в составе инструкции SELECT и отобразить выбранные строки на экране. Убедившись, что это именно те строки, которые требуется удалить, можно использовать предложение WHERE в инструкции на удаление.

Также возможна ситуация, когда поля удаляемых строк являются внешним ключом для какой-либо другой таблицы. В таком случае просто так удалить данные не получится, будет появляться ошибка. Для этих целей при создании таблиц в *Oracle* нужно использовать опции:

– ON DELETE SET NULL – в другой таблице удаляемые данные, которые являются внешним ключом на исходную таблицу, будут заменены на NULL;

– ON DELETE CASCADE – в другой таблице будут удалены строки, для которых удаляемые данные из исходной таблицы являются внешним ключом.

Оператор DELETE довольно долгий, так как все изменяемые данные заносятся в сегменты отката. Если нужно очистить всю таблицу целиком и при этом данные точно не понадобятся в дальнейшем, то можно использовать оператор TRUNCATE:

TRUNCATE TABLE *имя таблицы*.

Таким образом, с помощью TRUNCATE нельзя удалить часть строк, только все (в таблице или кластере), эту инструкцию нельзя откатить, и она не вызывает срабатывание триггеров.

Инструкция DELETE с подчиненным запросом. Инструкции DELETE с простыми условиями отбора отбирают строки для удаления исключительно на основании содержимого этих строк. Но иногда отбор строк необходимо производить опираясь на данные из других таблиц. Предположим, вы хотите удалить все объекты, принадлежащие какому-либо владельцу. Не зная его идентификатора, невозможно найти принадлежащие ему объекты. Чтобы найти эти объекты, надо обратиться с запросом к таблице владельцев (OWNER). Для решения подобных задач *SQL* предоставляет возможность задания подзапросов в предложении WHERE инструкции DELETE. Задача удаления информации об объектах, принадлежащих владельцу Иванову, решается следующим образом:

```
DELETE FROM property_for_rent
WHERE property_for_rent.ono IN (SELECT ono FROM owner
WHERE fname = 'Иванов');
```

Подчиненные запросы в предложении WHERE могут иметь несколько уровней вложенности. Они могут также содержать внешние ссылки на целевую таблицу инструкции DELETE. В этом случае предложение FROM инструкции DELETE играет такую же роль, как и предложение FROM инструкции SELECT.

3.13. Обновление данных (инструкция UPDATE)

Инструкция UPDATE обновляет значения одного или нескольких столбцов в выбранных строках одной таблицы.

UPDATE *имя_таблицы*
SET *имя_столбца*_1 = *значение*_1 [, *имя_столбца*_2 = *значение*_2 …]
[WHERE *условие*];

В инструкции указывается таблица, значения в которой должны быть изменены, при этом пользователь должен иметь разрешение на обновление таблицы и каждого конкретного столбца. Предложение WHERE отбирает строки таблицы, подлежащие обновлению. В предложении SET указывается, какие столбцы должны быть обновлены, и для них задаются новые значения. Приведем пример инструкции, увеличивающей заработную плату менеджеров на 10 %:

```
UPDATE staff
SET salary = salary*1.1
WHERE position = 'Manager';
```

Условия отбора, которые могут быть заданы в предложении WHERE инструкции UPDATE, в точности соответствуют условиям отбора, доступным в инструкциях SELECT и DELETE.

Как и инструкция DELETE, инструкция UPDATE может одновременно обновить несколько строк, соответствующих условию отбора. Например, необходимо перевести всех служащих из отделения 2 в отделение 1:

```
UPDATE staff
SET bno = 1
WHERE bno = 2;
```

Предложение SET в инструкции представляет собой список операций присваивания, отделяемых друг от друга запятыми. В каждой операции идентифицируется столбец, который должен обновляться, и определяется новое значение для этого столбца. Каждый целевой столбец должен встречаться в списке только один раз. Согласно стандарту *ANSI*/*ISO* для столбцов необходимо использовать простые имена, но некоторые СУБД допускают использование полных имен столбцов.

Присваиваемое значение может быть сформировано с помощью выражения. Выражение в операции присваивания может быть любым допустимым *SQL*-

выражением, результирующее значение которого имеет тип данных, соответствующий целевому столбцу. Необходимо, чтобы значение выражения вычислялось на основе значений строки, которая в данный момент обновляется в целевой таблице. Оно не может включать в себя какие-либо статистические функции или подчиненные запросы.

Если выражение в операции присваивания содержит ссылку на один из столбцов таблицы, то для вычисления выражения используется значение этого столбца в текущей строке, которое было перед обновлением. То же самое справедливо для ссылок на столбцы в предложении WHERE.

Обновление всех строк. Предложение WHERE в инструкции UPDATE является необязательным. Если оно опущено, то обновляются все строки целевой таблицы. Например, увеличить заработную плату всех сотрудников на 5 %:

```
UPDATE staff
SET salary=1.05*salary;
```

Инструкции UPDATE с подчиненным запросом. В инструкции UPDATE, так же как и в инструкции DELETE, подчиненные запросы могут играть важную роль, поскольку они дают возможность отбирать строки для обновления, опираясь на информацию из других таблиц. Например, всем служащим минского отделения увеличить зарплату на 10 %:

```
UPDATE staff
SET salary = 1.1*salary
WHERE bno IN (SELECT bno
FROM branch WHERE city = 'Минск');
```

Подчиненные запросы в предложении WHERE инструкции UPDATE, так же как и в инструкции DELETE, могут иметь любой уровень вложенности и содержать внешние ссылки на целевую таблицу инструкции UPDATE.

При использовании инструкций INSERT, UPDATE, DELETE в интерактивном режиме работы в *SQL*Plus* необходимо выполнять команду COMMIT для сохранения изменений в таблице, чтобы они сохранились и после окончания сессии. Также пока изменения не зафиксированы, остальные пользователи их не видят и можно вернуть все изменения обратно с помощью оператора ROLLBACK.

Контрольные вопросы и задания к разделу 3

1) Заполнить учебную БД данными, используя *DML*-инструкции (не менее пяти записей в каждой таблице).

2) Создать дополнительную таблицу-архив с данными об объектах недвижности типа «квартира» при помощи инструкции CREATE TABLE … SELECT.

3) Создать дополнительную таблицу с информацией о сотрудниках, заполнить ее с помощью инструкции MERGE.

4) С помощью многострочной инструкции вставки перенести информацию в таблицу *owner* из таблицы *staff*.

5) Вывести фамилии и телефоны арендаторов, в которых встречаются цифры 25 или 29. При выводе поля должны быть фиксированной длины (дополнить, например, точками), поле «Фамилия» выравнивается по левому краю, поле «Телефон» – по правому.

6) Определить и вывести с помощью запроса количество уникальных собственников, объекты которых уже осмотрели и на которые оставили комментарий.

7) Вывести информацию об отделениях, в которых работают более трех сотрудников.

8) Вывести информацию о сотрудниках отделений, в которых не обслуживается ни один объект недвижимости.

9) Сформулируйте и напишите запрос с использованием предиката IN с подзапросом.

10) Сформулируйте и напишите запрос с использованием предиката ANY/ALL с подзапросом.

11) Сформулируйте и напишите запрос с использованием предиката EXISTS/NOT EXISTS с подзапросом.

12) Выведите в одном столбце информацию обо всех сотрудниках, арендаторах и собственниках.

13) Обновите заработную плату сотрудников, увеличив ее на 10 % от среднего значения по всей компании, и только тем сотрудникам, у которых были осмотры объектов недвижимости в текущем месяце.

14) Удалите отделение с наименьшим количеством осмотров и всех его сотрудников.

15) Какие ограничения в синтаксисе имеют подчиненные запросы?

16) Чем различаются инструкции DELETE и TRUNCATE?

17) Какие соединения таблиц называют внешними?

18) Что понимают под естественным соединением таблиц, какой результат, например, оно будет возвращать для таблиц *staff* и *renter*?

19) Какие ключевые слова инструкции SELECT используют для работы с данными, имеющими иерархическую структуру.

20) Для чего и с каким синтаксисом используется предложение FETCH?

4. ТРАНЗАКЦИИ

4.1. Понятие транзакции

Как правило, изменения в БД обусловлены реальными событиями, происходящими в предметной области и требующими нескольких изменений таблиц. Например, при увольнении сотрудника требуется закрепить все объекты увольняемого сотрудника за другими сотрудниками отделения и удалить запись о сотруднике.

Чтобы не возникло противоречий с данными, указанные изменения следует выполнить как одно целое. Если из-за системного сбоя или другой ошибки получится, что одна часть изменений была внесена, а другая – нет, то это нарушит целостность хранимых данных. Поэтому несколько изменений БД, которые вызваны одним событием, необходимо вносить по принципу «либо все, либо ничего». Это обеспечивается средствами обработки механизма транзакций.

Транзакцией называют несколько последовательных инструкций *SQL*, которые рассматриваются СУБД как единое целое. Каждая инструкция решает часть общей задачи, но для того чтобы задачу можно было считать решенной, требуется выполнить все эти инструкции.

Понятие транзакции (рис. 4.1) является очень важным для программ, изменяющих содержимое БД, так как позволяет обеспечить ее целостность. Транзакции в реляционной СУБД подчиняются следующему правилу: инструкции, входящие в транзакцию, рассматриваются как неделимое целое, либо все инструкции будут выполнены успешно, либо ни одна из них не должна быть выполнена.

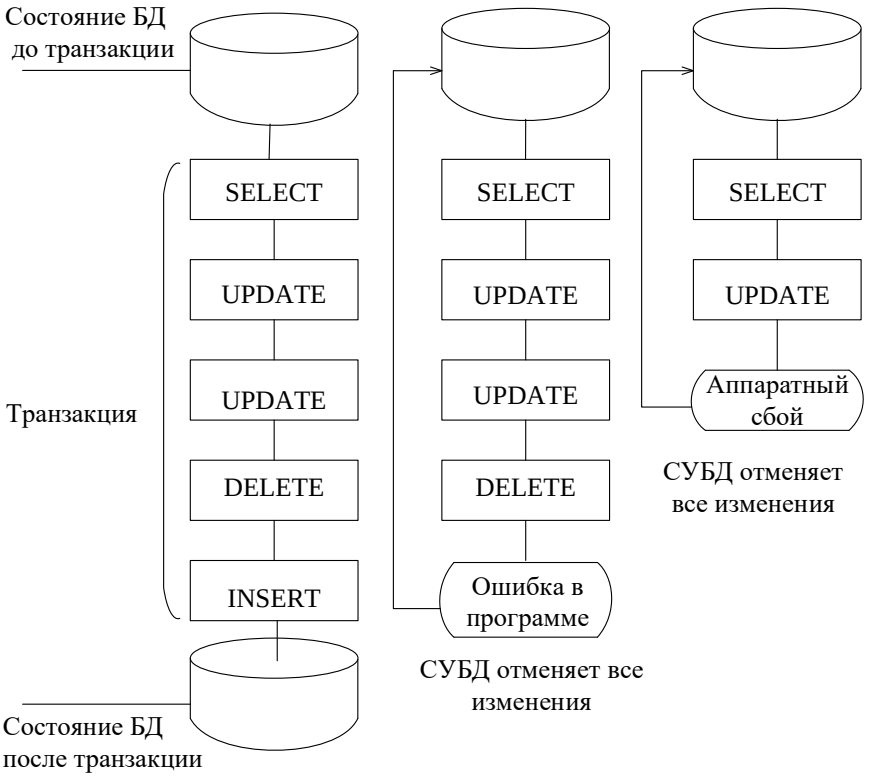


Рис. 4.1. Понятие транзакции

СУБД должна подчиняться этому правилу, даже если во время выполнения транзакции произойдет ошибка в программе или аппаратный сбой. В любом случае должно гарантироваться, что при восстановлении БД после сбоя «частичное выполнение транзакции» в ней отражено не будет.

Инструкции COMMIT и ROLLBACK. В СУБД обработка транзакций реализована с помощью двух инструкций:

– инструкция COMMIT сообщает об успешном окончании транзакции. Она информирует СУБД о том, что транзакция завершена, все инструкции, входящие в ее состав, выполнены успешно и противоречия в БД не возникли;

– инструкция ROLLBACK сообщает о неуспешном окончании транзакции. Она информирует СУБД о том, что пользователь не будет завершать транзакцию; все изменения, внесенные в базу данных в результате выполнения транзакции, должны быть отменены, т. е. СУБД должна возвратить БД в состояние, в котором она находилась до начала транзакции.

Инструкции COMMIT и ROLLBACK являются такими же инструкциями *SQL*, как и SELECT, INSERT или UPDATE. Однако в отличие от инструкций, изменяющих данные в базе, инструкции управления транзакциями редко употребляются в интерактивном режиме. При интерактивной обработке смысл их применения становится очевидным только в контексте параллельной работы с сетевой БД нескольких пользователей.

Проверка успешности выполнения инструкций, входящих в транзакцию, выполнение отката транзакции (ROLLBACK) или подтверждение ее успешного выполнения (COMMIT) осуществляется обычно программно, посредством специально разработанных и стандартизированных интерфейсов (например, *ODBC API – Open Database Connectivity, Application Programming Interface* – открытый доступ к БД, интерфейс прикладного программирования, который является отраслевым стандартом). В функцию интерфейсов помимо пересылки по сети пользовательских запросов входит также обеспечение корректного соединения с БД на основе существующих сетевых протоколов, а также предварительная обработка запросов, включающая их грамматический разбор и «связывание» параметров в предложениях WHERE.

Модель транзакции в стандарте *ANSI/ISO*. В стандарте *ANSI/ISO* определена модель транзакции, а также указаны задачи инструкций COMMIT и ROLLBACK. В большинстве коммерческих СУБД используется именно эта модель, которая была создана на основе модели, принятой в СУБД *DB*2. В стандарте говорится, что транзакция автоматически начинается с выполнения пользователем или программой первой инструкции *SQL*. Далее происходит последовательное выполнение остальных инструкций *SQL* до тех пор, пока транзакция не завершится одним из четырех способов.

– Инструкция COMMIT завершает выполнение текущей транзакции. Изменения, внесенные в БД, становятся постоянными. Новая транзакция начинается непосредственно после инструкции COMMIT.

– Инструкция ROLLBACK отменяет выполнение текущей транзакции. Произведенные изменения отменяются. Новая транзакция начинается непосредственно после инструкции ROLLBACK.

– Успешное завершение программы (для программного *SQL*) считается также и успешным окончанием транзакции, как если бы была выполнена инструкция COMMIT. Поскольку программа завершена, новая транзакция не начинается.

– Неуспешное завершение программы (для программного *SQL*) считается также и неуспешным окончанием транзакции, как если бы была выполнена инструкция ROLLBACK. Поскольку программа завершена, новая транзакция не начинается.

На рис. 4.2 в графическом виде представлены четыре перечисленные ситуации.

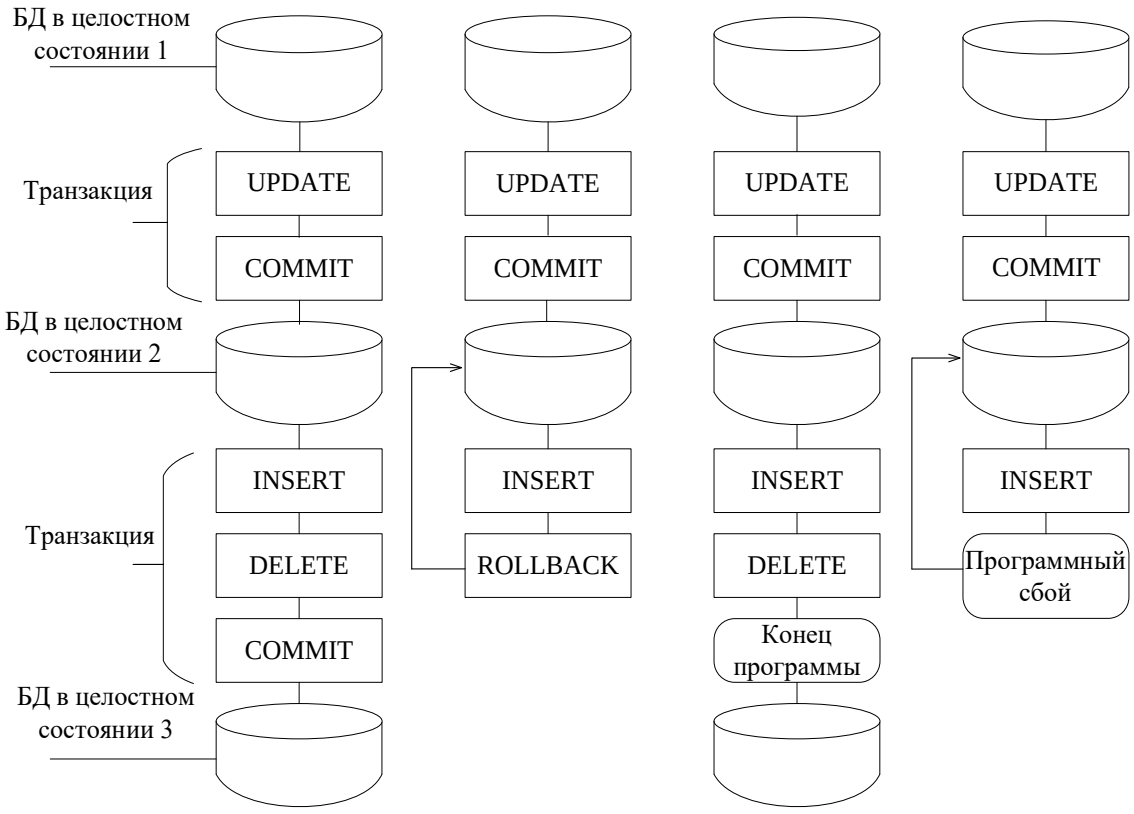


Рис. 4.2. Модель транзакции в стандарте *ANSI/ISO*

Отметим, что, согласно модели транзакции в стандарте *ANSI/ISO*, пользователь или программа могут выполнить транзакцию всегда. Чтобы начать транзакцию, не требуется предпринимать никаких специальных действий; транзакция начинается автоматически вместе с первой инструкцией *SQL* или непосредственно после окончания предыдущей транзакции.

В стандарте *ANSI/ISO* существует понятие *программного SQL*, т. е. *SQL*, предназначенного для использования в прикладных программах. В программном *SQL* транзакции играют важную роль, поскольку даже в простых прикладных программах для выполнения определенной задачи часто приходится применять

последовательность из двух или трех инструкций. В связи с тем что пользователь может изменить принятое решение или могут возникнуть какие-либо другие обстоятельства (например, на складе отсутствует продукция, которую пользователь хочет заказать), прикладная программа после частичного выполнения транзакции должна иметь возможность выбора: либо выполнить транзакцию до конца, либо отменить ее. Инструкции COMMIT и ROLLBACK обеспечивают именно такую возможность.

Как уже указывалось, инструкции COMMIT и ROLLBACK можно использовать и в интерактивном режиме, но на практике это случается редко. Обычно интерактивный режим применяется для запросов на выборку и гораздо реже – для запросов на изменение; изменения, состоящие из нескольких инструкций, практически никогда не вводятся в интерактивном режиме. На самом деле во многих интерактивных СУБД по умолчанию установлен режим «автоматического завершения», при котором инструкция COMMIT выполняется после каждой инструкции *SQL*, вводимой пользователем. В результате каждая инструкция *SQL* становится отдельной транзакцией. В СУБД *Oracle*, как уже отмечалось при рассмотрении *DML*-команд, только при работе в *SQL*Plus* необходимо указывать явно инструкцию COMMIT, иначе при завершении сеанса пользователя данные не будут изменены.

4.2. Журнал транзакций и проблемы параллельной обработки

Реализация в СУБД принципа «все или ничего» по отношению к инструкциям транзакции вызывает закономерный вопрос, каким образом СУБД может отменить изменения, внесенные в БД, если во время выполнения транзакции происходит системная ошибка? Для этого СУБД используются различные методы, но все они, как правило, основаны на применении *журнала транзакций*.

Когда пользователь выполняет запрос на изменение, СУБД автоматически вносит в журнал транзакций одну запись для каждой строки, измененной в процессе выполнения запроса. Эта запись содержит две копии строки. Одна копия представляет собой строку *до изменения*, а другая – *после изменения*. Только когда в журнале будет сделана запись, СУБД изменит физическую строку на диске. Затем, если пользователь выполняет инструкцию COMMIT, в журнале отмечается конец транзакции. Если пользователь выполняет инструкцию ROLLBACK, СУБД обращается к журналу и извлекает из него исходные копии строк, измененных во время транзакции. Используя эти копии, СУБД возвращает строки в прежнее состояние и таким образом отменяет изменения, внесенные в БД в ходе транзакции.

В общем случае при системном сбое администратор БД восстанавливает ее с помощью специальной утилиты восстановления, поставляемой вместе с СУБД. Эта утилита просматривает журнал транзакций и отыскивает транзакции, которые не были завершены к моменту сбоя и отменяет их. Таким образом, в БД будут отражены только завершенные транзакции.

На рис. 4.3 приведен упрощенный вид журнала транзакций.



Рис. 4.3. Журнал транзакций

Транзакции и работа в многопользовательском режиме. Если с БД одновременно работают двое или более пользователей, обработка транзакций приобретает новый смысл. Теперь СУБД должна не только осуществлять восстановление БД после отмены транзакции или системного сбоя, но и гарантировать, что пользователи не будут мешать друг другу, т. е. обеспечивать независимость работы пользователей друг от друга. В идеальном случае каждый пользователь должен работать с БД так, как если бы он имел к ней монопольный доступ, и не беспокоиться о действиях других пользователей. Средства обработки транзакций в *SQL* позволяют реляционным СУБД «изолировать» пользователей друг от друга именно таким образом.

Чтобы понять, как выполняются параллельные транзакции, следует рассмотреть проблемы, которые возникнут, если эти транзакции не обрабатывать положенным образом. Хотя такие проблемы довольно многочисленны, их можно разбить на четыре основные категории, рассматриваемые ниже.

1. Проблема пропавшего обновления. Рассмотрим проблему на примере параллельной работы хотя бы двух пользователей БД. На рис. 4.4 изображена схема работы прикладной программы, с помощью которой двое служащих принимают заказы от клиентов.

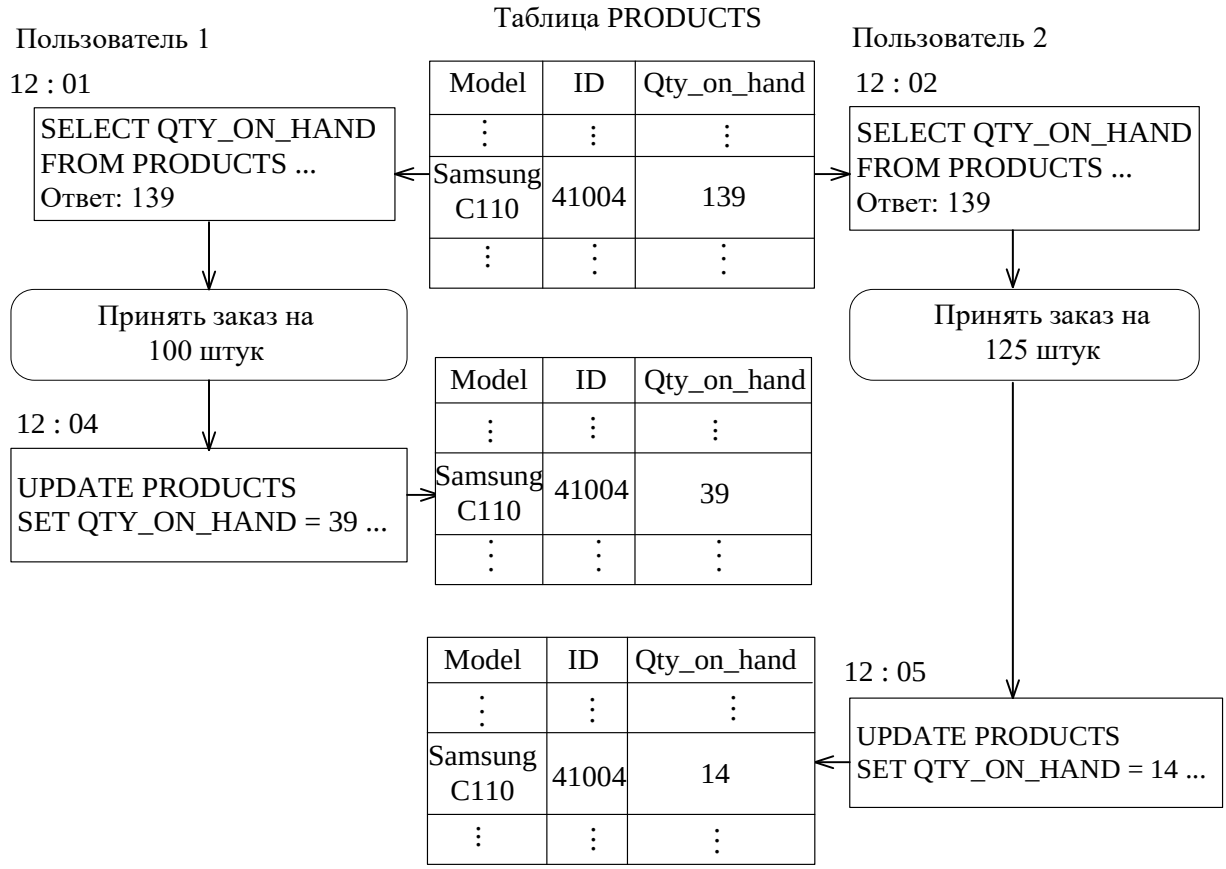


Рис. 4.4. Проблема пропавшего обновления

Перед вводом заказа программа по таблице PRODUCTS проверяет, имеется ли требуемое изделие в наличии. Пользователь 1 начинает вводить заказ от своего клиента на 100 изделий *Samsung C*110. В то же время Пользователь 2 начинает вводить заказ от своего клиента на 125 тех же изделий. Обе программы для ввода заказов выполняют запрос к таблице PRODUCTS, и каждая из них выясняет, что на складе имеется 139 требуемых изделий – количество, более чем достаточное для выполнения заказа. Пользователь 1 просит клиента подтвердить заказ; затем его копия программы обновляет таблицу PRODUCTS, показывая, что в наличии осталось 39 изделий *Samsung C*110, и добавляет в таблицу ORDERS новый заказ на 100 изделий. Через несколько секунд Пользователь 2 просит своего клиента подтвердить заказ. Его копия программы обновляет таблицу PRODUCTS, показывая, что на складе осталось 14 изделий *Samsung C*110, и добавляет в таблицу ORDERS новый заказ на 125 изделий.

Очевидно, что обработка этих двух заказов привела к возникновению противоречия в БД. Первое из двух обновлений таблицы PRODUCTS пропало. Заказы от обоих клиентов приняты, но на складе нет достаточного количества изделий для удовлетворения обоих заказов. Более того, таблица PRODUCTS показывает, что в наличии осталось еще 14 изделий. Пример показывает, что про-

блема пропавшего обновления может возникнуть всякий раз, когда две программы извлекают из базы одни и те же данные, используют их для каких-либо расчетов, а затем пытаются обновить эти данные.

2. Проблема промежуточных данных. На рис. 4.5 изображена схема работы той же программы для обработки заказов, что и на рис. 4.4.

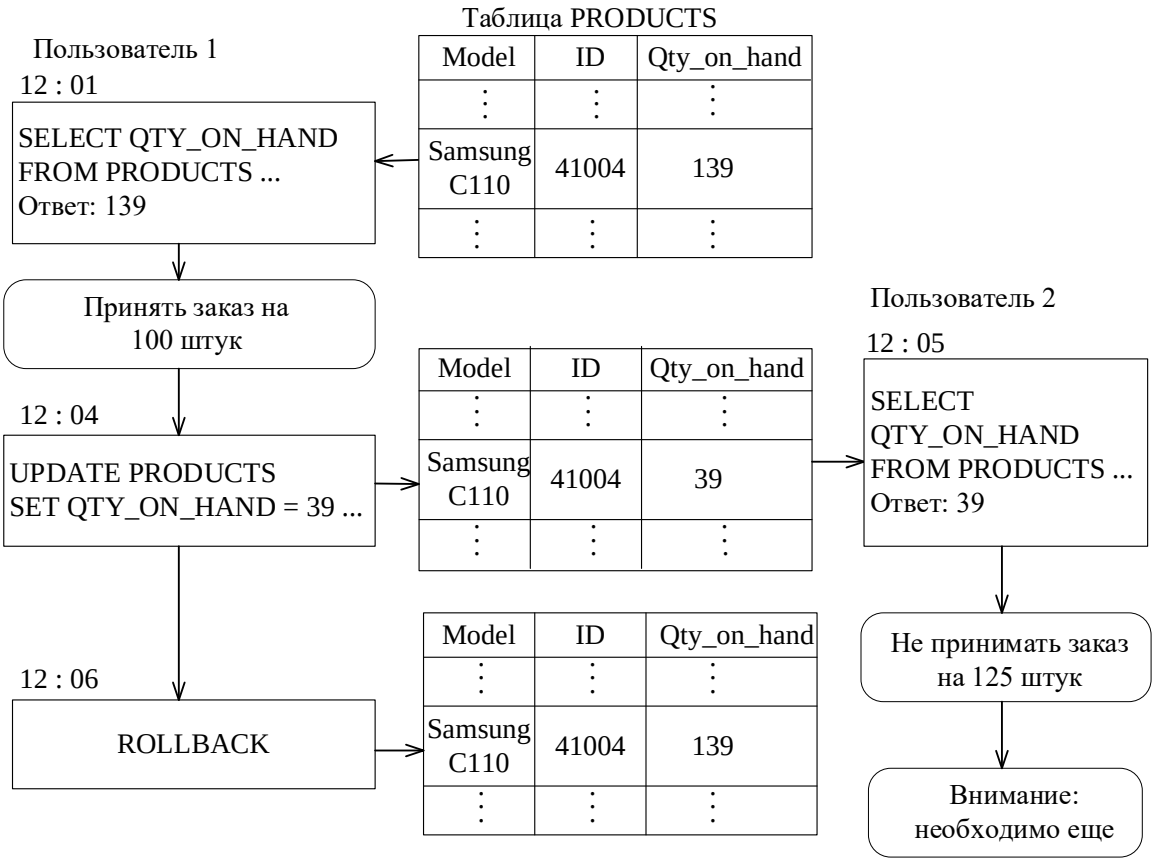


Рис. 4.5. Проблема промежуточных данных

На этот раз Пользователь 1 снова начинает принимать от клиента заказ на 100 изделий *Samsung C*110, его копия программы запрашивает таблицу PRODUCTS, выясняет, что в наличии имеется 139 изделий, и обновляет ее, показывая, что после принятия заказа в наличии осталось 39 изделий. Тем временем клиент Пользователя 2 пытается заказать 125 вышеупомянутых изделий. Его копия программы запрашивает таблицу PRODUCTS, выясняет, что в наличии имеется только 39 изделий, и отказывается принять заказ, а клиент Пользователя 1 решает отказаться от заказа, и программа выполняет инструкцию ROLLBACK для отмены транзакции, после выполнения которой СУБД восстанавливает в таблице число 139. Такое противоречие в базе данных возникло из-за того, что информация, которую извлекла программа Пользователя 2, являлась промежуточной, так как еще не была подтверждена программой Пользователя 1.

3. Проблема несогласованных данных. На рис. 4.6 изображена еще одна схема работы программы для обработки заказов.

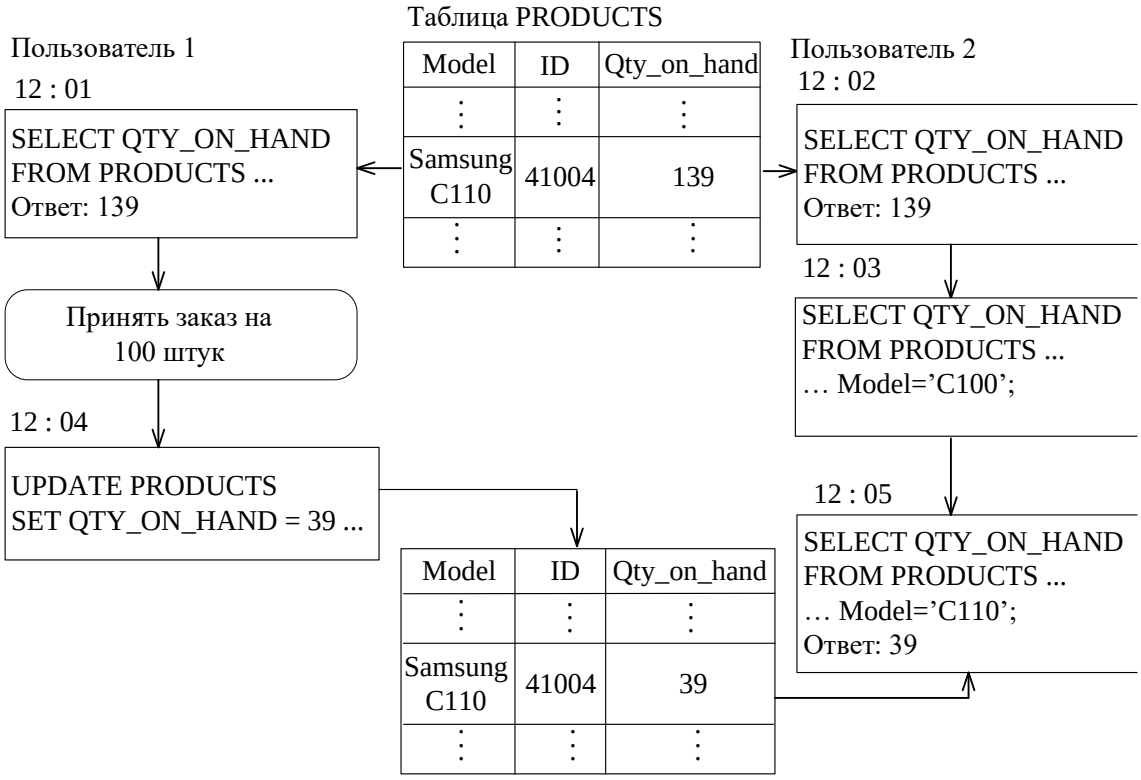


Рис. 4.6. Проблема несогласованных данных

Пользователь 1 начинает принимать от своего клиента заказ на 100 изделий *Samsung C*110. Вскоре после этого программа Пользователя 2 выясняет количество этих же изделий на складе. Затем программа Пользователя 2 выполняет запрос о наличии изделий *Samsung C*100. Тем временем Пользователь 1 принимает заказ на изделия *Samsung C*110, его программа обновляет соответствующую строку и выполняет инструкцию COMMIT, завершая транзакцию по приему заказа. После некоторых размышлений клиент Пользователя 2 решает заказать изделия *Samsung C*110, которые ему предлагались вначале. Его программа вновь запрашивает информацию об этих изделиях. Но новый запрос показывает, что в наличии имеется только 39 изделий вместо 139, показанных предыдущим запросом несколько секунд тому назад.

В данном примере, в отличие от двух предыдущих, состояние БД правильно отражает реальную ситуацию: в наличии осталось только 39 изделий. Из-за того что Пользователь 2 увидел данные программы Пользователя 1, ничего особенного не произошло – заказ от Пользователя 1 был успешно принят. Однако, с точки зрения Пользователя 2, БД не была целостной в течение выполняемой им транзакции. В начале транзакции некоторая строка содержала одни данные, а позднее в той же самой транзакции – другие, поскольку «внешние события» нарушили целостное состояние данных. Такая несогласованность может привести к различным проблемам даже в том случае, если программа Пользователя 2 не будет обновлять БД, опираясь на результаты первого запроса. Например, если его программа накапливает итоговые суммы или собирает статистиче-

ские данные, то нет гарантии, что она отражает информацию правильно. В стандарте эта проблема обозначена как «P2», или проблема «нестабильных результатов выборки».

4. Проблема строк-фантомов. На рис. 4.7 еще раз изображена схема работы программы для обработки заказов.

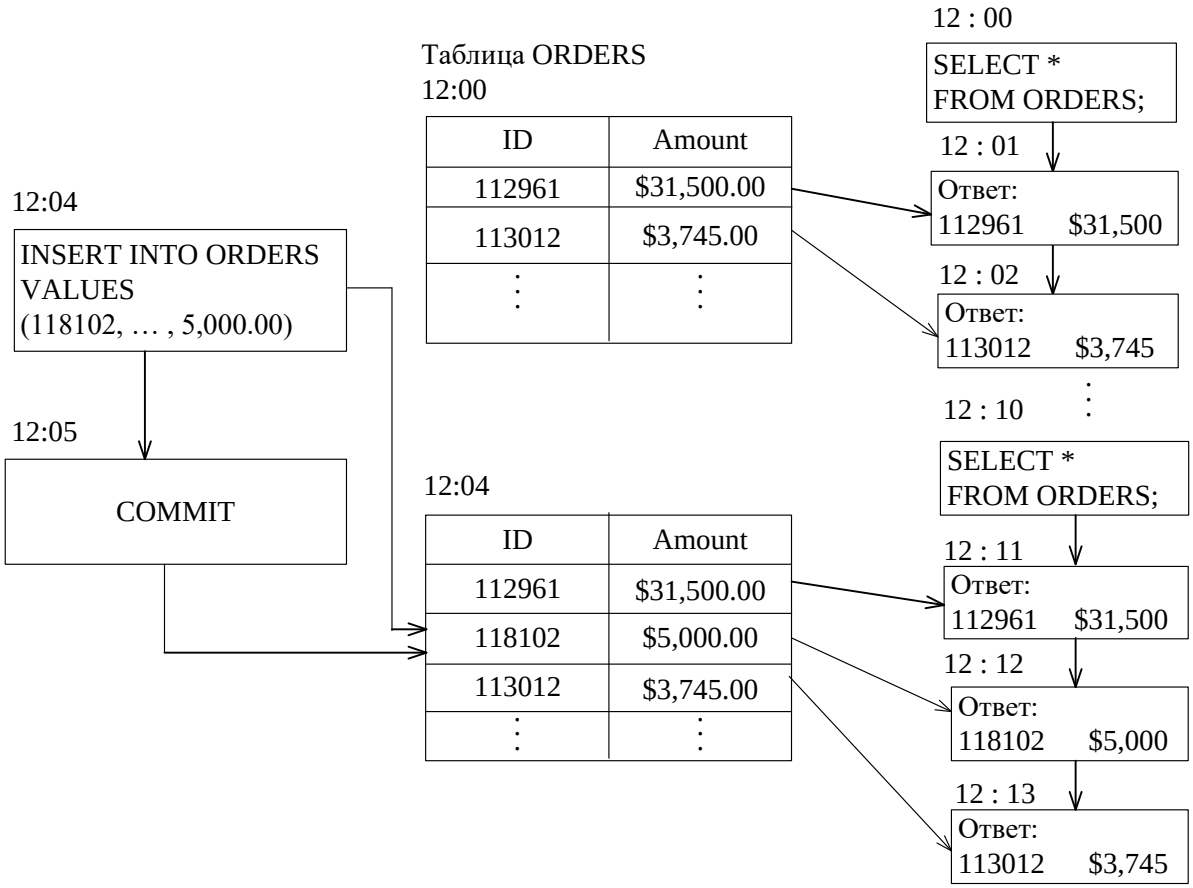


Рис. 4.7. Проблема строк-призраков

На этот раз менеджер запустил программу генерации отчетов, которая просматривает таблицу ORDERS и печатает список заказов от клиентов какого-то сотрудника, подсчитывая их итоговую сумму. Тем временем этот сотрудник принимает дополнительный заказ на $5000. Заказ добавляется в БД, и транзакция завершается. Вскоре после этого менеджер по продажам снова просматривает таблицу ORDERS, выполняя тот же запрос, что и прежде. На этот раз в таблице имеется дополнительный заказ, а итоговая сумма заказов на $5000 больше, чем в результате первого запроса.

Здесь, как и в предыдущем примере, проблема заключается в несогласованности данных. Состояние БД соответствует реальной ситуации, и целостность данных не нарушена, но один и тот же запрос, выполненный дважды в течение одной транзакции, возвращает два различных результата. В предыдущих примерах запрос извлекал одну строку, и противоречивость данных была вызвана выполнением инструкции UPDATE. Выполнение инструкции DELETE

могло бы вызвать ту же проблему. В данном примере проблема возникла в результате выполнения инструкции INSERT. В первом запросе дополнительной строки не было, и после выполнения второго запроса сложилось такое впечатление, что она появилась «из ниоткуда». Проблема строк-фантомов, как и проблема несогласованных данных, может привести к противоречивым и неправильным расчетам. В стандарте эта проблема обозначена как «РЗ».

Параллельные транзакции. Как видно из приведенных примеров, при обновлении БД в многопользовательском режиме существует возможность нарушения ее целостности. Чтобы исключить такую возможность, в *SQL* используется механизм транзакций. Помимо того что реляционная СУБД обязана выполнять транзакции по принципу «либо все, либо ничего», она имеет и другое обязательство:

«В ходе выполнения транзакции пользователь видит полностью непротиворечивую базу данных. Он не должен видеть промежуточные результаты транзакций других пользователей, и даже завершение таких транзакций не должно отражаться на данных, которые пользователь видит в течение транзакции».

Таким образом, в реляционной БД транзакции играют ключевую роль как при восстановлении после сбоев, так и при параллельной работе нескольких пользователей. Приведенное выше правило можно сформулировать иначе, пользуясь концепцией параллельного выполнения транзакций:

«Когда две транзакции, **A** и **B**, выполняются параллельно, СУБД гарантирует, что результаты их выполнения будут точно такими же, как если бы вначале выполнялась транзакция **A**, а затем транзакция **B**; или вначале выполнялась транзакция **B**, а затем транзакция **A**».

Данная концепция называется *сериализацией* транзакций. На практике это означает, что каждый пользователь может работать с БД так, как если бы не было других пользователей, работающих параллельно.

Однако тот факт, что СУБД изолирует пользователя от действий других пользователей, не означает, что о них можно забыть. Совсем наоборот. Поскольку другим пользователям также требуется параллельно обновлять БД, транзакции должны быть как можно более простыми и короткими, чтобы объем работы, выполняемой всеми пользователями, был максимальным.

Предположим, например, что вы запускаете программу, последовательно выполняющую три больших запроса на выборку. Так как программа не обновляет БД, может показаться, что нет необходимости использовать инструкцию COMMIT. Однако на самом деле программа должна выполнять эту инструкцию *после каждого* запроса, так как транзакция начинается *автоматически* вместе с первой инструкцией *SQL* в программе. Без инструкции COMMIT транзакция будет продолжаться до окончания программы. Кроме того, СУБД гарантирует, что данные, извлекаемые в течение транзакции, будут непротиворечивыми и не будут зависеть от транзакции других пользователей. Это означает, что если программа извлекла из БД определенную строку, *то ни один пользователь, кроме текущего, не сможет изменить эту строку до окончания данной транзакции.* Так происходит из-за того, что позднее в этой же транзакции можно снова из-

влечь ту же строку, а СУБД должна гарантировать, что в этой строке будут содержаться те же данные, что и при первой выборке. Поэтому, по мере того как программа будет последовательно выполнять три запроса, другие пользователи не смогут изменять все большее количество данных.

Поэтому при написании программ для реляционных БД всегда необходимо учитывать работу с транзакциями. Транзакции должны быть как можно короче, т. е. рекомендуется использовать инструкцию COMMIT как можно раньше и чаще – это хороший тон для всех, кто работает с программным *SQL*.

4.3. Блокировки транзакций

Практически во всех СУБД для обработки параллельных транзакций применяется довольно сложный механизм блокировки. Однако принципы блокировки транзакций довольно просты. На рис. 4.8 изображена схема простой блокировки, устраняющей конфликты между двумя параллельными транзакциями.

Когда транзакция **A** обращается к БД, СУБД автоматически блокирует все части БД, в которых транзакция осуществляет выборку или изменение. Транзакция **B** выполняется параллельно, и СУБД также блокирует те части БД, к которым она обращается. Если транзакция **B** обращается к той части БД, которая заблокирована транзакцией **A**, то СУБД приостанавливает выполнение транзакции **B**, заставляя ее ждать до тех пор, пока данные не будут разблокированы. СУБД снимает блокировку, вызванную транзакцией **A**, только после того, как в этой транзакции встретится инструкция COMMIT или ROLLBACK. Затем СУБД позволяет продолжить выполнение транзакции **B**. Теперь транзакция **B** блокирует эту же часть БД, защищая ее от других транзакций.

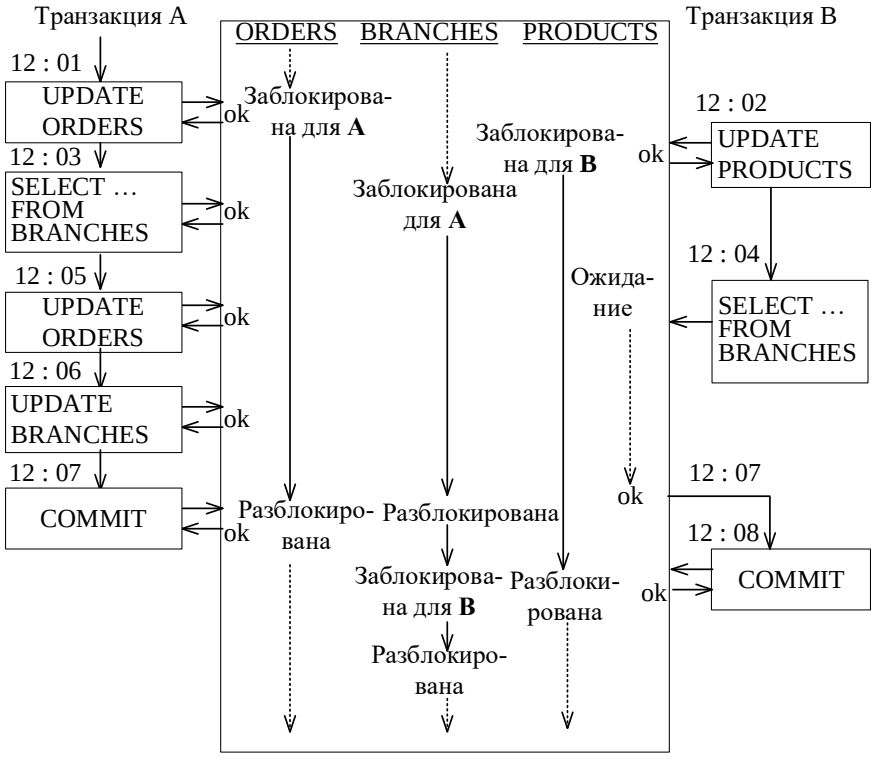


Рис. 4.8. Простая блокировка параллельных транзакций

Как видно из рис. 4.8, при блокировке транзакция временно получает монопольный доступ к некоторой части БД, причем другим транзакциям запрещается изменять заблокированные данные. Таким образом, блокировка решает все проблемы, возникающие при параллельном выполнении транзакций. Она не допускает разрушения БД в результате потери обновлений и использования промежуточных или несогласованных данных. Однако блокировка создает новую проблему: период времени, в течение которого транзакция ожидает освобождения части БД, заблокированной другой транзакцией, может быть достаточно большим.

Уровни блокировки. В БД блокировка может быть реализована на различных уровнях. Самой грубой формой блокировки является блокировка всей БД. Этот вид блокировки легко реализовать, но при этом в каждый момент времени можно будет выполнять только одну транзакцию. Если транзакция затрачивает некоторое время «на размышления» (например, время обсуждения заказа с клиентом), то доступ всех остальных пользователей к БД будет при этом заблокирован, что приводит к слишком низкой производительности.

Другой формой блокировки является блокировка на уровне таблиц. В этом случае СУБД блокирует только те таблицы, к которым обращается транзакция. Остальные транзакции в это время могут обращаться к другим таблицам. Этот вид блокировки предпочтительнее, чем блокировка БД, поскольку он позволяет проводить параллельную обработку транзакций. Но в таких приложениях, как программы для ввода заказов, в которых несколько пользователей одновременно обращаются к одним и тем же таблицам, данный вид блокировки также приводит к слишком низкой производительности.

Во многих СУБД реализована блокировка *на уровне страниц*. В этом случае СУБД блокирует отдельные блоки данных («страницы») на диске. Остальным транзакциям запрещается доступ к заблокированным страницам, но они могут обращаться к другим страницам данных (и блокировать их для себя). Обычно используются страницы размером 2, 4 и 16 Кбайт. Поскольку большая таблица состоит из множества страниц, две транзакции, обращающиеся к двум различным строкам таблицы, как правило, обращаются к различным страницам; в результате обе транзакции выполняются параллельно.

В большинстве ведущих СУБД реализована блокировка *на уровне строк*. Она допускает параллельное выполнение транзакций, которые обращаются к двум различным строкам таблицы, даже если эти строки находятся на одной странице. Хотя такая возможность кажется несущественной, в случае таблиц, содержащих небольшое число строк, она может играть важную роль.

Блокировка на уровне строк обеспечивает параллельное выполнение большого количества транзакций. К сожалению, осуществить блокировку строк различной длины гораздо сложнее, чем блокировку страниц фиксированного размера, поэтому повышение параллелизма работы обеспечивается за счет усложнения логики блокирования и роста накладных расходов.

Проблемы блокировки всегда являются предметом большого числа исследований. Сложность механизмов блокировки, используемых в коммерческих

СУБД, намного превышает сложность основных схем блокировки, описанных выше.

Жесткая и нежесткая блокировки. В большинстве коммерческих СУБД для повышения степени параллельности доступа нескольких пользователей к одной БД используются следующие типы блокировки:

– нежесткая блокировка, когда текущая транзакция извлекает информацию из БД и другие транзакции, выполняемые параллельно, могут извлекать те же данные;

– жесткая блокировка, когда текущая транзакция обновляет данные в базе, а другие транзакции не могут обращаться к ним ни для выборки, ни для записи.

В СУБД *Oracle* для определения жесткой и нежесткой блокировки используются термины **монопольная** и **разделяемая** блокировка.

На рис. 4.9 изображены допустимые комбинации блокировок для двух параллельно выполняемых транзакций. Следует отметить, что транзакция может применять для данных жесткую блокировку только в том случае, если ни одна другая транзакция не блокирует эти данные. Если транзакция пытается осуществить блокировку, не разрешенную правилами, ее выполнение приостанавливается до тех пор, пока другие транзакции не разблокируют необходимые ей данные.

		Транзакция **B**		
		Разблокирована	Нежесткая блокировка	Жесткая блокировка
Транзакция **A**	Разблокирована	ДА	ДА	ДА
	Нежесткая блокировка	ДА	ДА	НЕТ
	Жесткая блокировка	ДА	НЕТ	НЕТ

Рис. 4.9. Правила применения жесткой и нежесткой блокировок

Тупиковые ситуации. К сожалению, применение любого механизма блокировки для поддержки параллельного выполнения транзакций приводит к возникновению проблемы, которая называется *тупиковой ситуацией*. На рис. 4.10 изображена типичная тупиковая ситуация. Программа **A** обновляет таблицу ORDERS и поэтому блокирует часть этой таблицы. Тем временем программа **B** обновляет таблицу PRODUCTS и блокирует ее часть. Затем программа **A** пытается обновить таблицу PRODUCTS, а программа **B** пытается обновить таблицу ORDERS, причем каждая пытается обновить ту часть таблицы, которая заблокирована другой программой.

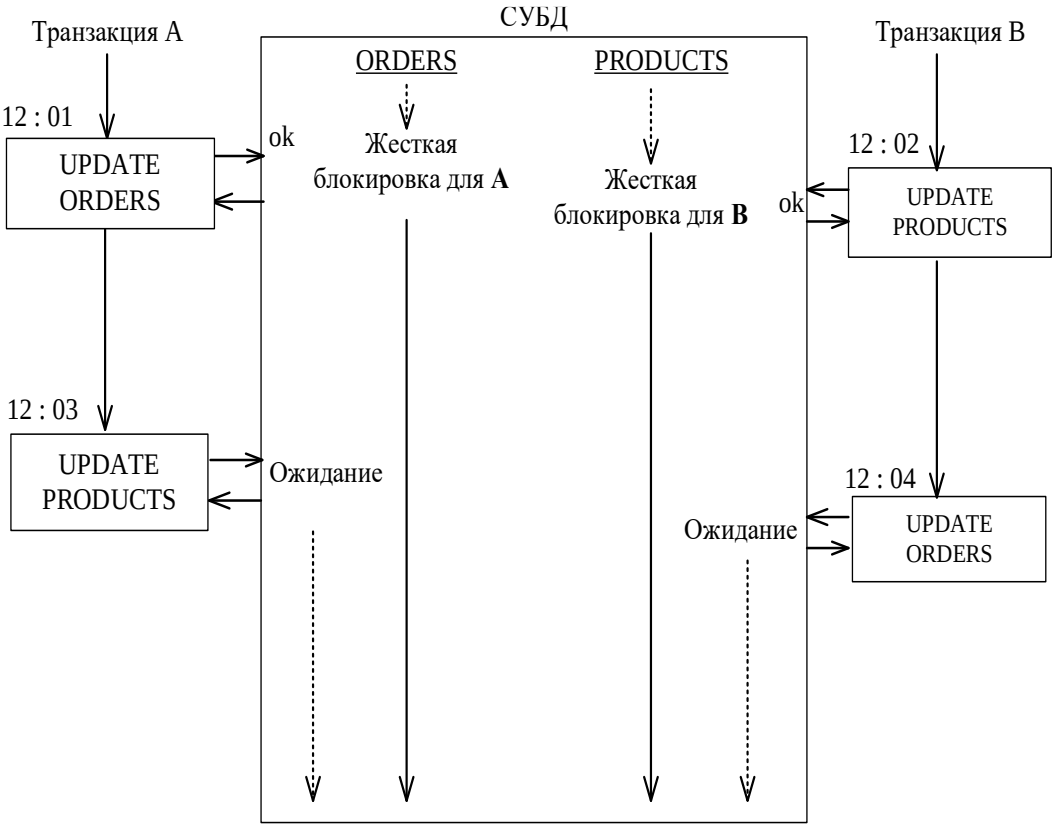


Рис. 4.10. Тупиковая ситуация

Без внешнего вмешательства обе программы будут бесконечно ожидать, пока другая программа завершит транзакцию и разблокирует данные. Но могут возникнуть и более сложные ситуации, когда три, четыре или даже больше программ попадают в «цикл» блокировок и каждая из них ожидает освобождения данных, заблокированных другой программой.

Как правило, для устранения тупиковых ситуаций СУБД периодически (например, каждые пять секунд) проверяет блокировки, удерживаемые различными транзакциями. Если СУБД обнаруживает тупик, то произвольно выбирает одну транзакцию в качестве «проигравшей» и отменяет ее. Это освобождает данные, удерживаемые проигравшей транзакцией, позволяя «победителю» продолжить работу. Проигравшей программе посылается код ошибки, показывающий, что она проиграла в тупиковой ситуации и ее текущая транзакция была отменена.

Такая схема устранения тупиков означает, что любая инструкция *SQL* может вернуть код ошибки «проигрыша в тупиковой ситуации», хотя сама по себе инструкция будет правильной. Транзакция, пытающаяся выполнить эту инструкцию, отменяется не по вине последней, а из-за параллельной работы с БД многих пользователей. Такая схема может показаться несправедливой, но на практике она лучше двух возможных альтернатив: «зависания» или нарушения целостности данных. Если ошибка «проигрыша в тупиковой ситуации» происходит в интерактивном режиме, пользователь может просто повторно ввести инструкции (одну или несколько). В программном *SQL* прикладная программа сама должна

обрабатывать код подобной ошибки и либо выдавать предупреждающее сообщение пользователю, либо пытаться выполнить транзакцию повторно.

Вероятность возникновения тупиков можно резко уменьшить, если тщательно планировать обновления БД. Все программы, которые в своих транзакциях обновляют несколько таблиц, по возможности должны всегда обновлять таблицы в одном и том же порядке. Это приводит к плавному перемещению блокировок по таблицам и сводит к минимуму возможность возникновения тупиков. Кроме того, для дальнейшего уменьшения количества возникающих тупиков можно использовать средства явной блокировки.

Явная блокировка. Если программа обращается к таблице многократно, затраты, связанные с блокировкой большого количества маленьких участков таблицы, могут оказаться значительными. Например, программа пакетного обновления, которая обращается к каждой строке таблицы, в процессе работы будет блокировать таблицу по частям. Для более быстрого выполнения транзакции такого типа программе следует явно заблокировать всю таблицу, выполнить все необходимые обновления, а затем разблокировать ее. Блокировка всей таблицы имеет три преимущества:

– устраняет затраты, связанные с блокировкой строк (страниц);

– исключает возможность того, что другая транзакция заблокирует часть таблицы, вынуждая программу пакетного обновления находиться в состоянии ожидания;

– исключает возможность того, что другая транзакция заблокирует часть таблицы и «загонит в тупик» программу пакетного обновления, вынуждая ее начать выполнение повторно.

Недостаток явной блокировки таблицы в том, что все остальные транзакции, которые пытаются обратиться к таблице, должны ждать окончания ее обновления. Но поскольку транзакция, выполняющая пакетное обновление, обычно работает гораздо быстрее остальных, общая производительность СУБД при явной блокировке таблиц может даже увеличиться.

Для явной блокировки целой таблицы применяется инструкция LOCK TABLE, стандартный синтаксис которой следующий:

LOCK TABLE *имя таблицы* IN SHARE|EXCLUSIVE MODE;

С помощью этой инструкции можно установить один из двух режимов блокировки:

– В режиме EXCLUSIVE осуществляется жесткая блокировка всей таблицы. В это время ни одна из других транзакций не имеет доступа ни к одной части таблицы. Этот режим следует использовать, если транзакция выполняет пакетное обновление всех строк таблицы.

– В режиме SHARE осуществляется нежесткая блокировка таблицы. Другие транзакции могут извлекать данные из таблицы, но не могут обновлять ее. Для того чтобы обновить часть таблицы, транзакция, выполнившая инструкцию LOCK TABLE, должна жестко заблокировать требуемую часть таблицы. Этот

режим блокировки следует использовать также в том случае, если требуется получить «мгновенный снимок» таблицы в какой-то определенный момент времени.

4.4. Понятие транзакции и виды блокировок в *Oracle*

В *Oracle* реализована уникальная модель поддержки конкурентного доступа, именуемая многоверсионной моделью согласованности по чтению (*Multiversion Read Consistency – MVRC*). Эта модель дает огромное преимущество для разработчиков, поскольку снимает с них большую часть забот о блокировках, возникающих в коде. Возможность поддержки *MVRC* встроена в самые глубины архитектуры СУБД *Oracle* [6].

Для осознания самой идеи конкурентного доступа и его реализации в СУБД *Oracle* необходимо знать несколько рассмотренных выше базовых понятий: транзакция, синтаксис команд COMMIT и ROLLBACK, блокировка, сериализация.

Рассмотрим альтернативные названия некоторых видов блокировок и проблем. Обычно для разрешения проблем в *Oracle* применяются блокировки двух типов.

Первый тип – это блокировка записи (*write lock*), или монопольная блокировка (*exclusive lock*). Монопольная блокировка применяется и удерживается в течение того времени, пока транзакция изменяет данные, а освобождается при завершении транзакции командой COMMIT или ROLLBACK. Блокировка записи предоставляет право на доступ к ресурсам единственному пользователю, который только и может изменять заблокированные данные.

В некоторых СУБД также применяется второй тип – блокировка чтения (*read locks*), или разделяемые блокировки (*shared locks*). Блокировка чтения может применяться любым количеством пользователей, которые просто читают данные, так как доступ только для чтения не создает конфликтных ситуаций. Однако блокировка чтения означает, что для данных нельзя будет применить блокировку записи, поскольку блокировка записи является монопольной. Обычно в *Oracle* применяются блокировки чтения, только если операция *SQL* специально запрашивает их в инструкции FOR UPDATE команды SELECT.

Несмотря на то что применение блокировок может привести к их конфликтам и ухудшению производительности, они все же необходимы для того, чтобы избежать перечисленных ниже проблем с нарушением целостности данных:

– «Грязное» чтение (*dirty reads*). Считывание данных называется «грязным», если СУБД разрешает одной транзакции читать данные, измененные другой транзакцией, когда эти изменения еще не были зафиксированы. Изменения, сделанные второй транзакцией, могут быть отменены, и тогда считанные данные окажутся некорректными. Многие СУБД разрешают «грязное» чтение для того, чтобы избежать конфликтов, создаваемых блокировками чтения.

– Невоспроизводимое чтение (*nonrepeat reads*). Речь идет о ситуации, когда данные изменяются другой транзакцией. Одна транзакция выполняет запрос

по какому-то определенному условию. После того как данные отправлены в транзакцию, но до того, как она зафиксирована, другая транзакция изменяет данные так, что некоторые из извлеченных ранее данных перестают удовлетворять условию выбора. Если бы запрос в первой транзакции был повторен еще раз, он вернул бы другое результирующее множество, поэтому любые изменения, выполненные на основе исходных результатов, могут уже не быть корректными. При повторном чтении уже прочитанных ранее данных в рамках одной и той же транзакции могут быть получены различающиеся результаты.

– Фантомное чтение (*phantom reads*). Такой вид чтения также вызван изменениями, выполненными другой транзакцией. Одна транзакция выполняет запрос по какому-то определенному условию. После того как данные отправлены в транзакцию, но до того, как она зафиксирована, другая транзакция вставляет в БД новые строки, которые удовлетворяют условию и могли бы быть выбраны первой транзакцией. Если транзакция выполняет изменения, считая, что условию удовлетворяют только строки, возвращенные запросом, то фантомное чтение может привести к получению ошибочных данных. Все данные, которые были прочитаны первым запросом, будут возвращены и вторым, но могут быть возвращены и дополнительные данные, поэтому может оказаться, что любые изменения, сделанные на основе исходных результатов, больше не являются корректными.

Полное решение задачи конкурентного доступа заключается в обеспечении наивысшего уровня изоляции для действий различных пользователей, обращающихся к одним и тем же данным. Согласно стандарту *SQL*92 высший уровень изоляции называется сериализацией (*serialization*). Сериализуемые транзакции характеризуются предсказуемостью и воспроизводимостью, а это две главных добродетели целостности данных.

Конечно, сделать так, чтобы сервер БД поддерживал работу тысяч пользователей и при этом каждый из них считал себя единственным, нелегкая задача, но модель *MVRC Oracle* ее решает.

4.5. Концепция многоверсионной модели согласованности по чтению

Модель конкурентного доступа *Oracle* или многоверсионная модель согласованности по чтению *(MVRC*) гарантирует, что пользователь всегда видит непротиворечивое представление запрошенных данных. Если другой пользователь изменяет запрошенные данные в процессе выполнения запроса, *Oracle* хранит версию данных в том виде, в каком они находились на момент начала выполнения запроса. Если на момент начала запроса какие-то транзакции уже начали выполняться, но еще не были зафиксированы, сервер *Oracle* обеспечивает недоступность для запроса изменений, внесенных такими транзакциями. Возвращенные в запрос данные будут отражать результаты выполнения всех транзакций, которые были зафиксированы на момент начала выполнения запроса.

Эта особенность *MVRC* влияет на производительность СУБД и на то, как запросы обращаются с БД. Основная концепция *MVRC* заключается в следующем.

1. *Oracle* не устанавливает на данные какие бы то ни было блокировки для операций чтения. Другими словами, операция чтения никогда не будет препятствовать операции записи. Даже если СУБД помещает одну-единственную блокировку всего на одну строку для операции чтения, это уже может привести к конфликтным ситуациям; к тому же для большей части таблиц БД операции обновления обычно производятся для небольшого количества «горячих точек» активных данных.

2. Операция записи никогда не будет препятствовать выполнению операции чтения. *MVRC* просто предоставит версию данных, которая существовала до начала выполнения операции записи.

3. Пользователь получает полный моментальный снимок данных точно на момент начала выполнения запроса. Строка, извлекаемая в конце результирующего множества, могла быть изменена за время получения всего результирующего множества. Но поскольку сервер *Oracle* хранит версию строки в том виде, в котором она существовала в момент начала запроса, вы всегда получаете непротиворечивую картину данных на конкретный момент времени.

4. Операция записи блокирует другую операцию записи, только если вторая пытается записать ту же самую строку.

Для реализации многоверсионной согласованности по чтению в сервере *Oracle* применяются три специальные структуры данных:

1. Сегменты отката (*rollback segments*) – это структуры СУБД *Oracle*, служащие для хранения информации, необходимой для возврата данных в исходное состояние в случае отката транзакции. Такая информация нужна для восстановления строк БД в состояние, в котором они находились в момент начала выполнения рассматриваемой транзакции. Начиная изменять данные в блоке, транзакция сначала записывает старый образ данных в сегмент отката. Хранящаяся в сегменте отката информация в основном используется в двух целях: для предоставления необходимых для отката транзакции сведений и для поддержки многоверсионной согласованности.

Сегмент отката – это не то же самое, что журнал, он находится в специальном табличном пространстве отката и содержит табличные данные. Журнал содержит только векторы изменений и предназначен для регистрации всех транзакций БД, необходимых для ее восстановления в случае сбоя системы, тогда как сегмент отката служит для обеспечения отката транзакций и согласованности чтения.

Блоки сегментов отката кешируются в *SGA* вместе с блоками таблиц и индексов. Если блоки сегментов отката долгое время не используются, то самые старые из них могут быть удалены из кеша и записаны на диск.

2. Системные номера изменений (*SCN*). Для того чтобы обеспечить целостность данных в БД, необходимо отслеживать порядок выполнения транзакций. Механизм, применяемый сервером *Oracle* для поддержания упорядоченности

транзакций во времени, называется системным номером изменения (*System Change Number – SCN*).

SCN – это логическая временная метка, предназначенная для отслеживания порядка транзакций. *Oracle* считывает метки *SCN* из журнала, когда требуется воспроизвести транзакции в их корректном первоначальном порядке. На основе меток *SCN Oracle* также определяет, когда можно удалять уже ненужную информацию из сегментов отката.

3. Блокировки в блоках данных. СУБД должна каким-то способом узнавать о том, что некоторая строка заблокирована. Большая часть СУБД хранит в памяти список блокировок, управляемых процессом *Lock Manager*. *Oracle* хранит блокировки с помощью индикатора в том блоке данных, в котором хранится строка. Для СУБД *Oracle* блок данных – это наименьший объем данных, который может быть прочитан с диска, поэтому при каждом запросе строки считывается блок, а внутри него возможна блокировка. Индикаторы блокировок хранятся в блоке данных, но при этом каждая блокировка действует только для отдельной строки из блока.

В дополнение к уже упомянутым структурам данных, которые непосредственно относятся к обеспечению многоверсионной согласованности чтения, *Oracle* характеризуется еще одной особенностью, позволяющей достичь большей степени параллелизма в условиях большого количества пользователей. Это нерасширяемые блокировки строк (*nonescalating locks*).

Для того чтобы снизить накладные расходы процесса управления блокировками, некоторые СУБД иногда распространяют блокировки на большее количество данных. Например, если какая-то определенная часть строк таблицы заблокирована, СУБД расширяет блокировку до целой таблицы, блокируя уже все строки таблицы, в том числе и те, которые не затрагиваются выполняемой командой *SQL*. Этот механизм уменьшает количество блокировок, которыми управляет процесс *Lock Manager*, но приводит к блокировке не затрагиваемых операцией строк. Но поскольку сервер *Oracle* хранит блокировку каждой строки в ее блоке данных, то нет необходимости в расширении блокировок – и *Oracle* никогда этого не делает.

Синтаксис *MVRC*. По большей части механизм *MVRC* работает «за кулисами», так что относящихся к нему синтаксических конструкций очень немного. Однако пользователям следует знать, какие существуют уровни изоляции, и уметь их устанавливать.

Уровень изоляции (*isolation level*) определяет, в какой мере операции одного пользователя изолированы от операций другого пользователя. В *Oracle* применяются два основных уровня изоляции, которые оказывают различное влияние на разработчиков и на пользователей.

Уровень READ COMMITTED вводит сериализацию на уровне команды. Иначе говоря, любой запрос получает непротиворечивое представление данных в том состоянии, в котором они существовали на момент начала запроса. Но

транзакция может включать в себя несколько команд, поэтому в рамках выполняемой транзакции может произойти невоспроизводимое или фантомное чтение. Данный уровень изоляции транзакций устанавливается *Oracle* по умолчанию.

Уровень SERIALIZABLE вводит сериализацию на уровне транзакций. Иными словами, любой запрос внутри транзакции получает непротиворечивое представление данных в том виде, в каком они существовали на момент начала транзакции.

Эти два уровня изоляции по-разному реагируют на транзакцию, которая при помощи блокировки запрошенной строки приостанавливает их работу. Оба уровня изоляции ждут снятия блокировки. Когда блокировка снята, операция, выполняемая с уровнем изоляции READ COMMITTED, просто повторяет попытку выполнения. Это вполне логичный подход для операций, которым важно только состояние данных на момент начала выполнения команды.

С другой стороны, если блокирующая транзакция фиксирует изменения данных, то операция, выполняющаяся с уровнем изоляции SERIALIZABLE, возвращает ошибку, указывающую на то, что сериализовать операции невозможно. Это также разумно, ведь после изменения блокирующей транзакцией состояния данных по сравнению с тем, каким оно было на момент начала транзакции SERIALIZABLE, операции записи для измененных строк становятся невозможны. В подобной ситуации программист добавляет в программу дополнительную логику, чтобы вернуться к началу транзакции SERIALIZABLE и начать ее заново.

Oracle поддерживает еще один дополнительный уровень изоляции: READ ONLY, когда пользователь может объявить для транзакции или сеанса уровень изоляции READ ONLY. Такой уровень явно запрещает операции записи. Как и уровень сериализации транзакций, READ ONLY обеспечивает точное представление данных на момент начала транзакции.

Уровни изоляции можно устанавливать при помощи команд *SQL*: ALTER SESSION – устанавливает уровень изоляции для сеанса; SET TRANSACTION – устанавливает уровень изоляции для транзакции.

Синтаксис команд следующий:

```
ALTER SESSION
SET ISOLATION_LEVEL = SERIALIZABLE | READ COMMITTED;
```

где ключевое слово SERIALIZABLE вводит сериализацию на уровне транзакций; а READ COMMITTED – на уровне команд (это значение установлено по умолчанию для *Oracle*).

```
SET TRANSACTION
ISOLATION LEVEL SERIALIZABLE | READ COMMITTED;
```

где SERIALIZABLE вводит сериализацию на уровне транзакции для данной транзакции, а READ COMMITTED – на уровне команды для данной транзакции и это значение также установлено по умолчанию для *Oracle*.

4.6. Управление транзакциями в *Oracle*

Модель транзакций реляционной СУБД *Oracle* основана на «единице работы» (*unit of work*) и поддерживает большую часть операций над транзакциями (нельзя использовать ROLLBACK FORCE). Транзакция неявно начинается с началом сеанса или при выполнении первой команды *SQL* после последней команды COMMIT или ROLLBACK (фактически при первом изменении данных). Транзакция заканчивается при выполнении команды COMMIT или ROLLBACK.

Транзакция не зависит от блоков *PL/SQL*. Транзакции могут охватывать несколько системных блоков или в одном блоке может быть несколько транзакций (автономных). *PL/SQL* поддерживает следующие команды для транзакций: COMMIT, ROLLBACK, SAVEPOINT, TRANSACTION и LOCK TABLE.

COMMIT делает изменения в БД постоянными и видимыми для других сеансов в соответствии со следующим синтаксисом:

COMMIT [WORK] [COMMENT *текст*];

где WORK – необязательное ключевое слово, применяется для улучшения читаемости и соответствия стандарту. Текст необязательного комментария COMMENT может иметь длину до 50 символов.

ROLLBACK отменяет незафиксированные изменения, сделанные в текущей транзакции, влияет до начала транзакции или до указанной точки сохранения.

ROLLBACK [WORK] [TO [SAVEPOINT] *имя_точки_сохранения*];

SAVEPOINT устанавливает точку сохранения (именованную точку обработки) для текущей транзакции. Установка точки транзакции делает возможным выполнение частичного отката.

SAVEPOINT *имя_точки_сохранения*;

где *имя_точки_сохранения* – это необъявленный идентификатор. Внутри транзакции может быть установлено несколько точек сохранения. Если повторно использовать имя точки сохранения, то точка передвинется на новую позицию и откат к исходной позиции данной точки сохранения будет невозможен.

SET TRANSACTION управляет типом транзакции.

SET TRANSACTION *тип_транзакции* NAME *имя*;

где *имя* – имя транзакции, доступное на протяжении ее выполнения; а *тип_транзакции м*ожет принимать следующие значения:

– READ ONLY – означает начало транзакции только для чтения. Указывает СУБД, что следует обеспечить целостность по чтению БД в пределах транзакции (по умолчанию для команды). Транзакцию заканчивают команды

COMMIT или ROLLBACK. Внутри такой транзакции разрешены только команды LOCK TABLE, SELECT, SELECT INTO, OPEN, FETCH, CLOSE, COMMIT и ROLLBACK. Попытка выполнения в транзакции только для чтения других команд, таких как INSERT или UPDATE, приводит к появлению ошибки ORA-1456;

– READ WRITE – обозначает начало транзакции READ WRITE; это тип по умолчанию;

– ISOLATION LEVEL SERIALIZABLE – аналогично транзакции READ ONLY обеспечивается целостность по чтению в пределах транзакции вместо поведения по умолчанию – целостности по чтению на уровне команды. Сериализуемые транзакции не позволяют изменять данные;

– ISOLATION LEVEL READ COMMITTED – если транзакции необходимы строки, заблокированные другими транзакциями, она будет ждать снятия блокировки;

– USE ROLLBACK SEGMENT *имя_сегмента_отката* – указывает СУБД, что следует использовать указанный сегмент отката. Полезна, когда лишь один сегмент отката имеет большой размер, и известно, что программе необходим большой сегмент отката (например, при операции закрытия в конце месяца).

LOCK TABLE обходит неявные блокировки БД на уровне строк, явно блокируя целиком одну или несколько таблиц в указанном режиме.

LOCK TABLE список_таблиц IN режим_блокировки MODE [NOWAIT];

где *список_таблиц* – список таблиц, разделенных запятыми; *режим_блокировки* – может принимать значения ROW SHARE (SHARE UPDATE), ROW EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE или EXCLUSIVE.

ROW SHARE разрешает конкурирующий доступ к заблокированной таблице, но запрещает пользователям блокировать всю таблицу для монопольного доступа (EXCLUSIVE). ROW SHARE и SHARE UPDATE это синонимы, используются для совместимости синтаксиса более ранних версий.

ROW EXCLUSIVE аналогично ROW SHARE, за исключением того, что дополнительно запрещает блокировать в режиме SHARE. ROW EXCLUSIVE применяется СУБД автоматически при операциях INSERT, UPDATE, DELETE.

SHARE разрешает одновременные запросы к таблице, но запрещает обновлять заблокированную таблицу.

SHARE ROW EXCLUSIVE используется для просмотра всей таблицы и разрешения пользователям просматривать строки в таблицы, но запрещает другие блокировки в режиме SHARE и запрещает обновлять строки.

EXCLUSIVE разрешает запросы к заблокированной таблице, но запрещает все остальные действия.

NOWAIT означает, что БД передает управление пользователю или программе данным разделом, подразделом или таблицей незамедлительно, даже

если на них наложена блокировка другим пользователем. В этом случае БД возвращает предупреждающее сообщение о том, что другим пользователем уже была наложена блокировка.

Другими словами, NOWAIT указывает, что СУБД не должна ждать освобождения блокировки, в то время как по умолчанию при встрече блокировки СУБД бесконечно ждет ее освобождения.

Контрольные вопросы и задания к разделу 4

1. Что представляет собой журнал транзакций?
2. Дайте определение транзакции.
3. Что такое сериализация транзакций?
4. Какие уровни блокировок поддерживаются в СУБД?
5. Какие инструкции *SQL* управляют транзакциями?
6. Какой вид блокировки реализуется командой LOCK TABLE?
7. Какие проблемы обработки параллельных транзакций существуют?
8. Для чего предназначена инструкция SET TRANSACTION?
9. Что такое *MVRC*?
10. Когда в *Oracle* происходит автоматическая блокировка записи данных?
11. Какие структуры данных применяются для реализации *MVRC*?
12. В каких случаях происходит откат транзакций?
13. Приведите пример тупиковой ситуации для БД агентства недвижимости?
14. Какие уровни изоляции поддерживаются в *Oracle*?
15. Для чего и где используется ключевое слово NOWAIT?
16. Запишите синтаксис инструкции ALTER SESSION, применяемый для управления транзакциями в СУБД *Oracle.*

5. СПОСОБЫ ОБЕСПЕЧЕНИЯ БЕЗОПАСНОСТИ ДАННЫХ

5.1. Принципы защиты данных в стандарте *SQL*

За реализацию системы безопасности отвечает СУБД. Язык *SQL* является фундаментом системы безопасности реляционной СУБД: требования, предъявляемые к системе защиты информации в БД, формулируются с помощью инструкций *SQL*. С защитой данных в *SQL* связаны три основные концепции:

– Действующими лицами в БД являются пользователи. Каждый раз, когда СУБД извлекает, вставляет, удаляет или обновляет данные, она делает это от имени какого-то пользователя. СУБД выполнит требуемое действие или откажется его выполнять в зависимости от того, какой пользователь запрашивает это действие.

– Объекты БД являются теми элементами, чья защита может осуществляться посредством *SQL*. Обычно обеспечивается защита таблиц и представлений. Большинству пользователей разрешается использовать одни объекты БД и запрещается использовать другие.

– Привилегии – это права пользователя на проведение тех или иных действий над определенным объектом БД. Например, пользователю может быть разрешено извлекать строки из некоторой таблицы и добавлять их в нее, но запрещено удалять или обновлять строки этой таблицы. У другого пользователя может быть другой набор привилегий.

Пользователи. Каждому пользователю в многопользовательской реляционной базе администратором данных присваивается идентификатор – краткое имя, однозначно определяющее пользователя для СУБД. Эти идентификаторы являются основой системы безопасности. Каждая инструкция *SQL* выполняется в СУБД от имени конкретного пользователя. От его идентификатора зависит, будет ли разрешено или запрещено выполнение инструкции. В БД на персональном компьютере может быть только один идентификатор пользователя, который обозначает пользователя, создавшего БД и являющегося ее владельцем.

На практике ограничения на имена идентификаторов зависят от реализации СУБД. Стандарт *SQL*1 позволял идентификатору пользователя иметь длину до 18 символов и требовал, чтобы он был допустимым *SQL*-именем. В некоторых СУБД для мэйнфреймов длина идентификаторов ограничивалась восемью символами. В *Sybase* и *SQL Server* идентификатор пользователя может иметь до 30 символов. Если важна переносимость, то лучше всего, чтобы у идентификатора пользователя было не более восьми символов. Следует иметь в виду также и то, что всем пользователям в одном определенном отделе можно присвоить *один и тот же* идентификатор, так как все они имеют *идентичные привилегии*.

В стандарте *ANSI*/*ISO* вместо термина «идентификатор пользователя» (*user-id*) употребляется термин «*идентификатор прав доступа*» (*authorization-id*), и он встречается иногда в документации по *SQL*. С технической точки зрения второй термин является более правильным, так как в действительности идентификатор определяет права или привилегии. Бывают ситуации, когда имеет смысл

присвоить нескольким пользователям одинаковый идентификатор. В других ситуациях один пользователь может пользоваться двумя или тремя различными идентификаторами. В корпоративной БД идентификатор прав доступа может относиться к программам и группам программ, а не к отдельным людям.

Аутентификация пользователей. Безопасность БД обеспечивается с помощью идентификаторов пользователей. Однако в стандарте ничего не говорится о механизме связи идентификатора пользователя с инструкциями *SQL*. Например, на сервере БД может существовать программа генерации отчетов, запланированная на выполнение каждый вечер, каков в этом случае должен быть идентификатор, если нет самого пользователя? Наконец, как обрабатываются запросы к БД по сети, если на локальном компьютере у пользователя может быть один идентификатор, а на удаленном компьютере – другой?

В большинстве коммерческих реляционных СУБД идентификатор пользователя создается для каждого сеанса связи с БД. В интерактивном режиме сеанс начинается, когда пользователь запускает программу формирования запросов, и продолжается до тех пор, пока пользователь не выйдет из программы. В приложении, использующем программный *SQL*, сеанс начинается, когда приложение подключается к СУБД, и заканчивается по завершении работы приложения. В течение сеанса все инструкции *SQL* ассоциируются с идентификатором пользователя, установленным для этого сеанса.

Как правило, в начале сеанса необходимо ввести как идентификатор пользователя, так и связанный с ним пароль, который служит для подтверждения того, что пользователь действительно имеет право работать под введенным идентификатором. Идентификаторы и пароли применяются в большинстве реляционных СУБД, однако способ, которым пользователь вводит свой идентификатор и пароль, меняется в зависимости от СУБД.

В некоторых СУБД реализована своя собственная система безопасности с идентификаторами пользователей и паролями. Например, когда в СУБД *Oracle* клиентский доступ осуществляется через *SQL*Plus,* то вводятся имя пользователя и соответствующий пароль.

Защищаемые объекты. Средства защиты в *SQL* применяются по отношению к отдельным объектам БД. В стандарте *SQL*1 указаны два типа защищаемых объектов – таблицы и представления. Таким образом, каждая таблица и представление могут быть защищены индивидуально. Доступ к ним может быть разрешен для одних пользователей и запрещен для других. Стандарт *SQL*2 уже расширяет круг защищаемых объектов, включая в него домены, и пользовательские наборы символов.

В большинстве коммерческих реляционных СУБД дополнительно могут быть защищены и другие объекты. Например, в *Oracle* важным объектом БД является хранимая процедура. С помощью средств защиты *SQL* устанавливается, какие пользователи могут создавать и удалять хранимые процедуры и какие пользователи могут их выполнять.

Привилегии. Как уже указывалось, привилегии – это права пользователя на проведение тех или иных действий над определенным объектом БД. В *SQL*

предоставление тех или иных прав над различными объектами БД возможно посредством инструкции GRANT. Обычно инструкцией GRANT пользуется владелец таблицы или представления, чтобы разрешить другим пользователям доступ к ним. Многие коммерческие СУБД поддерживают набор системных привилегий. Системная привилегия – это мощная привилегия, которая предоставляет пользователю возможность выполнять системную операцию определенного вида. СУБД *Oracle* поддерживает и системные и объектные привилегии.

Работа с привилегиями при помощи ролей. Вполне возможно, что для использования обычного приложения БД придется назначать множество системных привилегий. Когда с приложением работает множество пользователей, управление привилегиями может быстро превратиться в достаточно трудную задачу, так как нужно будет предоставлять привилегии каждому пользователю. Чтобы упростить процесс обеспечения безопасности системы, некоторые СУБД, в частности *Oracle*, позволяют воспользоваться ролями. Роль – это совокупность системных привилегий, предоставляемых пользователям и другим ролям. Например, при создании нового приложения можно создать новую роль с привилегиями, необходимыми для выполнения данной программы. После того как пользователю приложения предоставляется эта роль, он может запускать приложение, соединяться с БД и выполнять свою работу. Если привилегии, необходимые для выполнения приложения, изменяются, то нужно только быстро модифицировать набор привилегий, заданных в роли. Все пользователи, которым предоставлена эта роль, автоматически отслеживают происшедшие изменения и продолжают работу с приложением, имея на то необходимые привилегии. СУБД может предоставлять пользователям одновременно некоторое количество предварительно установленных ролей.

Роли, определяемые пользователями. Для БД *Oracle* можно создавать любое требуемое количество ролей. После создания роли нужно построить для нее набор привилегий, предоставив ей привилегии и другие роли. Затем надо предоставить эту роль пользователям, чтобы они имели привилегии, необходимые им для работы.

Ниже приведен алгоритм создания новой роли для типового приложения по вводу заказов, а также предоставления роли ряду пользователей БД.

1. Необходимо создать роль:

```
CREATE ROLE order_entry;
```

2. Выдать роли *order_entry* необходимые привилегии:

```
GRANT SELECT, INSERT, UPDATE, DELETE ON sales.customers TO order_entry;
GRANT SELECT, INSERT, UPDATE, DELETE ON sales.orders TO order_entry;
GRANT SELECT, INSERT, UPDATE, DELETE ON sales.items TO order_entry;
GRANT SELECT, UPDATE ON sales.parts TO order_entry;
```

3. Выдать роль пользователям:

GRANT order_entry TO smith, alice, george;

В приведенном примере *sales* – это схема БД; *customers, orders, items, parts* – таблицы этой схемы; *smith, alice, george* – пользователи базы данных, полномочия которых распространяются на объекты схемы *sales*.

5.2. Разрешение и запрещение ролей в стандарте *SQL*

Пользователь, которому предоставлена роль, не всегда имеет доступ к ее привилегиям. В *Oracle* приложения могут разрешать и запрещать роли для любого пользователя. После того как приложение *разрешает* роль для пользователя, ему становятся доступны ее привилегии. И, соответственно, когда приложение *запрещает* роль для пользователя, он более не имеет доступа к ее привилегиям. Возможность динамически управлять набором привилегий позволяет приложениям обеспечивать постоянную корректность наборов привилегий пользователей во время работы с данным приложением. Например, когда пользователь запускает приложение по вводу заказов, с помощью *SQL*-команды SET ROLE это приложение разрешает ему применять для работы роль *order_entry*. Когда пользователь завершает свою работу, приложение запрещает для него эту роль и он не может применять привилегии приложения по вводу заказов во время работы с другим приложением.

Предоставление привилегий (инструкция GRANT). Инструкция GRANT применяется для предоставления привилегий в отношении поименованных объектов БД указанным пользователям. Обычно его использует владелец таблицы с целью предоставления доступа к ней другим пользователям. Инструкция GRANT в стандарте имеет следующий синтаксис:

GRANT {*привилегия* [,...] | ALL PRIVILEGES}
ON *имя_объекта* TO {*идентификатор_пользователя* [,...]| PUBLIC}
[WITH GRANT OPTION];

В качестве параметра *привилегия* может использоваться следующая синтаксическая конструкция:

{SELECT | DELETE | INSERT [(*имя_столбца*[,...])] | UPDATE [(*имя_столбца* [,...])]} | REFERENCES [(*имя_столбца*[,...])] | USAGE }

Ключевое слово ALL PRIVILEGES позволит предоставить указанному пользователю все существующие привилегии без их перечисления.

Ключевое слово PUBLIC означает предоставление доступа указанного типа не только всем существующим пользователям, но и всем тем, кто будет определен в БД впоследствии.

Параметр *имя_объекта* может быть именем таблицы БД, представлением, доменом, набором символов, ограничением и т. д.

Опция WITH GRANT OPTION дает право передавать все предоставленные им в отношении указанного объекта привилегии другим пользователям, которые, в свою очередь, будут наделены точно таким же правом передачи своих полномочий. Если данный параметр не будет указан, получатель привилегии не сможет передать свои права другим пользователям. Таким образом, владелец объекта может четко контролировать, кто получил право доступа к объекту и какие полномочия ему предоставлены.

Рассмотрим ситуацию, когда владелец передает другим пользователям право предоставлять привилегии на свой объект, более подробно. Пусть:

GRANT SELECT ON orders TO secretary WITH GRANT OPTION;

Теперь *secretary* может выполнить следующую инструкцию GRANT:

GRANT SELECT ON orders TO somebody;

Такая инструкция позволяет *somebody* извлекать данные из представления *orders*. Так как *secretary* не включил в свою инструкцию GRANT предложение WITH GRANT OPTION, цепочка заканчивается на *somebody*; он может извлекать данные из представления *orders*, но не может давать право на доступ к нему другому пользователю.

Отмена привилегий (инструкция REVOKE). В большинстве реляционных БД привилегии, предоставленные посредством инструкции GRANT, могут быть отобраны с помощью инструкции REVOKE. Инструкция REVOKE, имеющая структуру, аналогичную структуре инструкции GRANT, содержит набор отбираемых привилегий, объект, к которому относятся эти привилегии, и идентификаторы пользователей, у которых отбираются привилегии.

Инструкция REVOKE имеет следующий формат:

REVOKE [GRANT OPTION FOR] {*привилегия*[,...] | ALL PRIVILEGES}
ON *имя_объекта* FROM {*идентификатор_пользователя* [,..]| PUBLIC}
[RESTRICT | CASCADE];

Ключевое слово ALL PRIVILEGES означает, что для указанного пользователя отменяются все привилегии, предоставленные ему ранее. Необязательная фраза GRANT OPTION FOR позволяет для всех привилегий, переданных в исходной инструкции GRANT фразой WITH GRANT OPTION, отменять возможность их передачи независимо от самих привилегий.

Если указано ключевое слово RESTRICT, успешное выполнение команды REVOKE будет возможно лишь в том случае, когда перечисленные привилегии в инструкции GRANT не могут послужить причиной появления у каких-либо

других пользователей так называемых оставленных привилегий. С помощью параметра CASCADE удаляются все привилегии, которые иначе могли бы остаться у других пользователей. «Оставленными» являются привилегии, сохранившиеся у пользователя, которому они в свое время были предоставлены с помощью WITH GRANT OPTION.

Поскольку наличие привилегии необходимо для создания определенных объектов, вместе с ее удалением можно лишиться права, за счет использования которого был создан тот или иной объект (подобные объекты называются «брошенными»). Если в результате выполнения оператора REVOKE могут появиться брошенные объекты (например, представления), право будет отменено при условии, что в нем не указывается ключевое слово CASCADE. Если ключевое слово CASCADE в операторе присутствует, то для любых «брошенных» объектов, возникающих при выполнении исходного оператора REVOKE, будут автоматически выполнены команды DROP.

Привилегии, которые были предоставлены указанному пользователю другими пользователями, не могут быть затронуты оператором REVOKE. Следовательно, если другой пользователь также предоставил данному пользователю удаляемую привилегию, то право доступа к соответствующей таблице у указанного пользователя сохранится. Например, пусть пользователь A и пользователь E имели право INSERT на таблицу *product*. Пользователь A предоставил пользователю B привилегию INSERT для таблицы *product*, причем с указанием WITH GRANT OPTION (этап 1). Пользователь B передал эту привилегию пользователю C (этап 2). Затем пользователь C получил ее же от пользователя E (этап 3). Далее пользователь C предоставил упомянутую привилегию пользователю D (этап 4). Когда пользователь A отменяет привилегию INSERT для пользователя B, она не может быть отменена и для пользователя C, поскольку ранее он уже получил ее от пользователя E. Если бы пользователь E не предоставил данной привилегии пользователю C, то удаление привилегии пользователя B имело бы следствием каскадное удаление привилегий для пользователей C и D.

Таким образом, посредством инструкции REVOKE можно отобрать только те привилегии, которые вы предоставили ранее некоторому пользователю. У него могут быть также привилегии, предоставленные другими пользователями; ваша инструкция REVOKE не влияет на эти привилегии.

Также при использовании инструкции REVOKE следует учитывать, что если два разных пользователя предоставляют третьему пользователю одну и ту же привилегию на один и тот же объект, а затем один из них отменяет привилегию, то вторая привилегия остается в силе и по-прежнему разрешает пользователю доступ к объекту. Такую ситуацию в литературе иногда называют *перекрытием привилегий*.

5.3. Аутентификация в *Oracle*

СУБД *Oracle* предлагает большой выбор возможностей, позволяющих исключить несанкционированный доступ к БД и помогающих защитить данные от

просмотра и изменения неавторизованными пользователями. Основными концепциями безопасности *Oracle* являются аутентификация, профили, привилегии, роли и аудит [5].

Аутентификацией называется процесс распознавания авторизованных пользователей. По существу, процедура аутентификации позволяет системе убедиться, что пользователь действительно является тем, за кого себя выдает. В основе безопасности *Oracle* лежит концепция индивидуальной авторизации пользователей.

Системные пользователи. При установке *Oracle* всегда создаются два пользователя БД:

- SYS,
- SYSTEM.

Схема SYS содержит базовые таблицы и представления словаря данных, которые не рекомендуется изменять ни при каких обстоятельствах. Пользователь SYS получает роль DBA. По умолчанию пользователю SYS назначается пароль CHANGE_ON_INSTALL.

Пользователь SYSTEM нужен для создания дополнительных таблиц и представлений, содержащих административную информацию. Этот пользователь также получает роль DBA. По умолчанию ему назначается пароль MANAGER.

Для ограничения доступа к обширным возможностям этих пользователей можно при создании БД *Oracle* командой CREATE DATABASE применять инструкции USER SYS IDENTIFIED BY *пароль* и USER SYSTEM IDENTIFIED BY *пароль*.

Первый шаг на пути к аутентификации – это создание пользователей. Новые пользователи создаются командой CREATE USER, а свойства существующих пользователей можно изменять командой ALTER USER.

Синтаксис команды CREATE USER, которая создает пользователя и определяет его основные характеристики, приводится ниже:

CREATE USER *имя_пользователя*
IDENTIFIED {BY *пароль* | EXTERNALLY | GLOBALLY AS '*внешнее_имя*'}
[DEFAULT TABLESPACE *имя_табличного_пространства*]
[TEMPORARY TABLESPACE *имя_табличного_пространства*]
[QUOTA {*целое_число* (K | M) | UNLIMITED}ON *имя_табличного_пространства*]
[QUOTA {*целое_число* (K | M) | UNLIMITED} ON *имя_табличного_пространства* …]
[PROFILE *имя_профиля*]
[PASSWORD EXPIRE]
[ACCOUNT LOCK | UNLOCK]

В инструкции используются ключевые слова, указанные в табл. 5.1.

Таблица 5.1

Ключевые слова инструкции CREATE USER

Ключевое слово	Назначение
IDENTIFIED BY	Определяет один из трех способов аутентификации пользователя PASSWORD, EXTERNALLY или GLOBALLY AS
PASSWORD	Пользователь идентифицируется с помощью хранимого локально пароля, который должен состоять из однобайтных символов, принадлежащих набору символов БД
EXTERNALLY	Пользователь идентифицируется внешней службой, в частности операционной системой. Если пользователь должен получить доступ согласно его учетной записи в операционной системе, его имя должно начинаться значением параметра OS_AUTHENT_PREFIX
GLOBALLY AS	Пользователь идентифицируется службой каталогов предприятия. Значение *внешнее_имя* может содержать характерное имя (*Distinguished Name*), имеющееся в каталоге, или пустую строку, означающую, что каталог отобразит пользователя на соответствующую схему БД
DEFAULT TABLESPACE	Указывает табличное пространство, в которое по умолчанию помещаются объекты, создаваемые пользователем. По умолчанию это табличное пространство USER
TEMPORARY TABLESPACE	Указывает табличное пространство, отведенное для хранения временных сегментов пользователя. По умолчанию это табличное пространство TEMP
QUOTA	Указывает размер области, доступной пользователю в указанном табличном пространстве. Можно указать несколько инструкций *QUOTA* для нескольких табличных пространств. Допускается указание в килобайтах (K) или в мегабайтах (M). Значение UNLIMITED снимает ограничения по расходованию пространства пользователем
PROFILE	Указывает профиль, назначенный пользователю
PASSWORD EXPIRE	Указывает на то, что пользователю или администратору БД необходимо изменить пароль, прежде чем пользователь сможет получить доступ к БД
ACCOUNT LOCK | UNLOCK	Разрешает и запрещает доступ для данной учетной записи

Синтаксис команды ALTER USER:

ALTER USER *имя_пользователя*
[IDENTIFIED {BY *пароль* [REPLACE *старый_пароль*]
EXTERNALLY | GLOBALLY AS '*внешнее_имя*'}]
[DEFAULT TABLESPACE *имя_табличного_пространства*]
[TEMPORARY TABLESPACE *имя_табличного_пространства*]
[QUOTA {*целое_число* (K | M) | UNLIMITED} ON *имя_табличного_пространства*]
[QUOTA {*целое_число* (K | M) | UNLIMITED} ON *имя_табличного_пространства* …]
[PROFILE *имя_профиля*]
[DEFAULT ROLE {[*имя_роли*[,*имя_роли* …] |
ALL {EXCEPT [*имя_роли*[,*имя_роли* …]} | NONE
[PASSWORD EXPIRE]
[ACCOUNT LOCK | UNLOCK]
[*имя_пользователя* [, *имя_пользователя* … *инструкция_прокси_сервера*]

Большинство ключевых слов в команде ALTER USER имеют то же значение, что и в команде CREATE USER. Ключевые слова, указанные в табл. 5.2, применяются только в инструкции ALTER USER.

Таблица 5.2

Ключевые слова инструкции ALTER USER

Ключевое слово	Назначение
REPLACE *старый_пароль*	Если включена функция проверки сложности пароля, то при изменении пароля необходимо указать его старое значение
DEFAULT ROLE	Механизм ролей позволяет управлять группами привилегий для групп пользователей. Можно выдать пользователю несколько ролей, все роли (ALL), все, кроме перечисленных (ALL EXEPT), или не выдать ни одной
инструкция_ прокси_сервера	Имеет свой синтаксис, разрешает или запрещает соединение через прокси, может относиться к нескольким именам пользователей

Для удаления пользователя из БД *Oracle* и удаления всех объектов, принадлежащих этому пользователю, используется оператор DROP USER. Синтаксис оператора:

DROP USER *имя_пользователя* [CASCADE];

где *имя_пользователя* – имя пользователя, которого нужно удалить из БД *Oracle*;

CASCADE – необязательный параметр, который необходимо указать, если *имя_пользователя* владеет любыми объектами (таблицами или представлениями в своей схеме), которые также будут удалены, и все ограничения ссылочной целостности на объекты пользователя также будут сняты.

5.4. Профили в *Oracle*

Для того чтобы ограничить доступ пользователя к ресурсам или указать условие обработки паролей пользователя, можно установить ему *профиль* (*profile*). Ограничивая объем вычислительных ресурсов, доступных пользователю, можно предотвратить их перерасход и нанесение ущерба работе других пользователей, как в случае с так называемым *отказом в обслуживании* (*denial of service*). Наложение ограничений на администрирование паролей способствует защите процесса аутентификации в БД *Oracle*. Для того чтобы использовать профили, следует разрешить динамические ограничения ресурсов при помощи параметра инициализации RESOURCE_LIMIT или команды ALTER SYSTEM SET. Определив профиль командой CREATE PROFILE, можно назначить его пользователю посредством команды CREATE USER или ALTER USER.

Команда CREATE PROFILE позволяет создать профиль и назначить для него различные типы ограничений на расходование ресурсов. Для роли может быть указано несколько параметров. Инструкция имеет следующий синтаксис:

CREATE PROFILE *имя_профиля* LIMIT {*параметр_ресурса* | *параметр_пароля*};

Ключевые слова, указанные в табл. 5.3, применяются для всех параметров.

Таблица 5.3

Общие ключевые слова инструкции CREATE PROFILE

Ключевое слово	Назначение
UNLIMITED	Указывает, что для данного параметра нет ограничений
DEFAULT	Указывает, что параметр принимает значение, заданное для профиля DEFAULT. Изначально все значения профиля DEFAULT устанавливаются в UNLIMITED. Значения для профиля можно изменять при помощи команды ALTER PROFILE. Значения параметров ресурсов задаются целыми числами, а значения параметров пароля – выражениями

Ключевые слова, указанные в табл. 5.4, применяются для параметров ресурсов.

Таблица 5.4

Ключевые слова параметров ресурсов

Ключевое слово	Назначение
SESSIONS_PER_USER	Ограничивает количество одновременных сеансов пользователя
CPU_PER_SESSION	Ограничивает время процессора для пользовательского сеанса (в сотых долях секунды)
CPU_PER_CALL	Ограничивает время процессора для отдельного пользовательского вызова (в сотых долях секунды)

Окончание табл. 5.4

Ключевое слово	Назначение
CONNECT_TIME	Ограничивает общую фактическую продолжительность сеанса (в минутах). Если пользователь превышает значение этого параметра, сервер откатывает текущую транзакцию и завершает сеанс. Следующий вызов пользователя возвращает ошибку
IDLE_TIME	Ограничивает время непрерывного ожидания пользователя (в минутах). Ограничение времени ожидания не применяется для длительных запросов
LOGICAL_READS_PER_SESSION	Ограничивает количество логических блоков данных, считываемых в пользовательском сеансе как из памяти, так и с диска
LOGICAL_READS_PER_CALL	Ограничивает количество логических блоков данных, считываемых в каждом пользовательском вызове как из памяти, так и с диска
COMPOSITE_LIMIT	Ограничивает общую стоимость ресурсов сеанса (в сервисных единицах)

Ключевые слова, указанные в табл. 5.5, используются для параметров пароля.

Таблица 5.5

Ключевые слова параметров пароля

Ключевое слово	Назначение
FAILED_LOGIN_ATTEMPTS	Ограничивает количество неудачных попыток регистрации пользователя, предшествующих блокировке его учетной записи
PASSWORD_LIFE_TIME	Устанавливает ограничение максимального срока действия одного пароля (в днях) для пользователя. По истечении срока пароль теряет силу
PASSWORD_REUSE_TIME	Указывает, сколько дней должно пройти, прежде чем пароль можно будет задать повторно. Если для параметра задано целочисленное значение, то необходимо установить PASSWORD_ REUSE_ MAX в UNLIMITED
PASSWORD_REUSE_MAX	Указывает, сколько раз необходимо изменить пароль, прежде чем будет разрешено задать его повторно. Если для параметра задано целочисленное значение, то необходимо установить PASSWORD_REUSE_TIME в UNLIMITED

Окончание табл. 5.5

Ключевое слово	Назначение
PASSWORD_LOCK_TIME	Указывает количество дней, на которое будет заблокирована учетная запись пользователя в случае превышения количества попыток неудачной регистрации
PASSWORD_GRACE_TIME	Указывает, за какое количество дней до истечения срока действия пароля выдается предупреждение
PASSWORD_VERIFY_ FUNCTION *функция* | NULL | DEFAULT	Разрешает применение функции *PL/SQL* для проверки сложности пароля

Команда ALTER PROFILE изменяет ограничения на расходование ресурсов или ограничения пароля для существующего профиля:

ALTER PROFILE *имя_профиля*
LIMIT {*параметр_ресурса* | *параметр_пароля*};

В команде ALTER PROFILE применяются те же ключевые слова и значения, что и в CREATE PROFILE (см. табл. 5.3–5.5).

Существующий профиль при необходимости можно удалить командой DROP PROFILE:

DROP PROFILE *имя_профиля* (CASCADE);

Ключевое слово CASCADE *у*казывает, что назначение профиля для всех активных пользователей должно быть отменено, а их профили необходимо заменить на DEFAULT. Данную инструкцию следует применять при удалении профиля текущего активного пользователя.

5.5. Привилегии в *Oracle*

Привилегии – это права, назначаемые отдельным пользователям или ролям. Привилегии в *Oracle* делятся на два основных вида:

– *системные* – дают пользователю или роли возможность выполнять определенные системные операции;

– *объектные* – дают пользователю или роли права доступа к отдельным объектам схемы.

Системные привилегии относятся к экземпляру *Oracle* в целом, например, есть привилегия для всех объектов одного типа. Объектные привилегии связаны с конкретным объектом схемы внутри БД *Oracle*.

Системные привилегии. Перечислим все системные привилегии *Oracle* в табл. 5.6.

Таблица 5.6

Общие системные привилегии

Привилегия	Назначение
ANY	Дает привилегию на выполнение операции над объектами любой схемы. В отсутствие такого ключевого слова привилегия выдается только на объекты в рамках схемы пользователя. По умолчанию ключевое слово ANY дает пользователю привилегию на все объекты всех схем, включая SYS. Для того чтобы запретить привилегии ANY на доступ к схеме SYS, можно установить параметр инициализации 07_DISTIONARY_ACCESSIBILITY в FALSE
ALTER	Дает привилегию на изменение объекта некоторого типа
CREATE	Дает привилегию на создание объекта данного типа
DROP	Дает привилегию на удаление объекта данного типа
EXECUTE	Дает привилегию на исполнение объекта данного типа или обращение к нему
SELECT ANY	Дает привилегию на доступ к объектам, применяется только с ключевым словом ANY, потому что пользователь всегда имеет возможность доступа к объектам собственной схемы

Для примера ниже приведены некоторые из системных привилегий, существующих в *Oracle*:

– CREATE SESSION (установить сеанс) – позволяет пользователю соединяться с сервером БД и устанавливать с ней сеанс связи;

– CREATE TABLE (создать таблицу) – позволяет пользователю создавать таблицы в своей собственной схеме;

– CREATE ANY TABLE (создать любую таблицу) – позволяет пользователю создавать таблицы в любой схеме БД;

– CREATE ANY TYPE (создать любой тип) – позволяет пользователю создавать типы и тела соответствующих типов в любой схеме БД;

– SELECT ANY TABLE (выбрать любую таблицу) – позволяет пользователю обращаться с запросами к любой таблице БД;

– EXECUTE ANY PROCEDURE (выполнить любую процедуру) – позволяет пользователю выполнять любую хранимую процедуру, хранимую функцию или модульный компонент БД.

Каждая из представленных разновидностей привилегий является общей и может применяться со многими типами уникальных системных привилегий, описанных в табл. 5.7.

Таблица 5.7

Уникальные системные привилегии

Объект привилегий	Описание привилегий	Уникальные привилегии	Общие привилегии
AUDIT	Разрешает функции аудита	AUDIT SYSTEM	ANY
CLUSTER	Предоставляет возможность работы с кластерами	–	CREATE [ANY], ALTER ANY, DROP ANY
CONTEXT	Предоставляет возможность работы с контекстами	–	CREATE ANY, DROP ANY
DATABASE	Предоставляет возможность выполнения команды ALTER DATABASE	–	ALTER
DATABASE LINKS	Предоставляет возможность работы со связями БД	CREATE PUBLIC, DROP PUBLIC	CREATE
DEBUG	Предоставляет возможность работы с отладчиком	DEBUG CONNECT, SESSION	–
DIMENSION	Предоставляет возможность работы с измерениями	–	CREATE [ANY], ALTER ANY, DROP ANY
DIRECTORY	Предоставляет возможность работы с каталогами	–	CREATE ANY, DROP ANY
INDEX	Предоставляет возможность работы с индексами	QUERY REWRITE	CREATE ANY, ALTER ANY, DROP ANY
INDEXTYPE	Предоставляет возможность работы с объектами типа INDEXTYPE	–	CREATE [ANY], ALTER ANY, DROP ANY, EXECUTE ANY
LIBRARY	Предоставляет возможность работы с библиотеками	–	CREATE [ANY], DROP ANY
MATERIALIZED VIEW	Предоставляет возможность работы с материализованными представлениями	ON COMMIT REFRESH	CREATE [ANY], ALTER ANY, DROP ANY

Продолжение табл. 5.7

Объект привилегий	Описание привилегий	Уникальные привилегии	Общие привилегии
OPERATOR	Предоставляет возможность работы с определяемыми пользователем операторами	–	CREATE [ANY], DROP, EXECUTE
OUTLINE	Предоставляет возможность работы с хранимыми планами выполнения	SELECT ANY	CREATE ANY, ALTER ANY, DROP ANY
PROCEDURE	Предоставляет возможность работы с процедурами	–	CREATE [ANY], ALTER ANY, DROP ANY, EXECUTE ANY
PROFILE	Предоставляет возможность работы с профилями	–	CREATE, ALTER, DROP
RESOURCE COST	Предоставляет возможность присваивания стоимостей ресурсам	–	ALTER
ROLE	Предоставляет возможность работы с ролями	GRANT ANY	CREATE, ALTER ANY, DROP ANY
ROLLBACK SEGMENT	Предоставляет возможность работы с сегментами отката	–	CREATE, ALTER, DROP
SEQUENCE	Предоставляет возможность работы с последовательностями	–	CREATE [ANY], ALTER ANY, DROP ANY, SELECT ANY
SESSION	Предоставляет возможность работы с сеансами	ALTER RESOURCE COST, RESTRICTED SESSION	CREATE, ALTER
SYNONYM	Предоставляет возможность работы с синонимами	–	CREATE [ANY] [PUBLIC], DROP ANY, DROP PUBLIC
SYSTEM	Предоставляет возможность изменения параметров системы	–	ALTER

Окончание табл. 5.7

Объект привилегий	Описание привилегий	Уникальные привилегии	Общие привилегии
TABLE	Предоставляет возможность работы с таблицами	BACKUP ANY, COMMENT ANY, INSERT ANY, LOCK ANY, FLASHBACK ANY, UPDATE ANY	CREATE [ANY], CREATE ALTER ANY, DELETE ANY, DROP ANY, SELECT ANY
TABLESPACE	Предоставляет возможность работы с табличными пространствами	MANAGE, UNLIMITED TABLESPACE	CREATE, ALTER, DROP
TYPE	Предоставляет возможность работы с определяемыми пользователем типами	UNDER ANY	CREATE [ANY], ALTER ANY, DROP ANY, EXECUTE ANY
USER	Предоставляет возможность работы с пользователями БД	BECOME	CREATE, ALTER, DROP
VIEW	Предоставляет возможность работы с представлениями	UNDER ANY	CREATE [ANY], DROP

Особые системные привилегии. Существуют также системные привилегии, предназначенные для предоставления пользователю возможности выполнения всего набора операций. Это особые привилегии, поскольку одна привилегия предоставляет пользователю набор базовых полномочий. Ознакомимся с ними в табл. 5.8

Таблица 5.8

Особые системные привилегии

Привилегия	Назначение	Полномочия
SYSDBA	Дает пользователю все права, необходимые для запуска и остановки экземпляра сервера БД	ALTER DATABASE CREATE DATABASE ARCHIVELOG, RECOVERY CREATE SPFILE
SYSOPER	Предоставляет пользователю чуть более ограниченный набор прав, предназначенный для оператора системы	ALTER DATABASE OPEN | MOUNT | BACKUP ARCHIVELOG, RECOVERY CREATE SPFILE

Привилегии доступа к объектам схемы. Существует несколько разновидностей привилегий доступа к объектам схемы. Они могут применяться к различным типам объектов схемы.

Общие привилегии доступа к объектам схемы приведены в табл. 5.9.

Таблица 5.9

Общие привилегии доступа к объектам схемы

Привилегия	Назначение
ALTER	Изменяет определение объекта
DEBUG	Обращается к *PL*/*SQL*-коду или информации о командах *SQL*, которые обращаются к объекту напрямую через отладчик
DELETE	Удаляет строки из объекта
EXECUTE	Компилирует или исполняет процедуру или функцию объекта либо обращается к программному объекту, объявленному в объекте
FLASHBACK	Выполняет ретроспективный запрос к объекту
INSERT	Добавляет в объект новые строки
REFERENCES	Создает ограничение (*constraint*), ссылающееся на объект
SELECT	Запрашивает объект
UNDER	Создает дочерний объект ниже уровня объекта
UPDATE	Изменяет существующие данные объекта

Объекты схемы и их уникальные привилегии. В разделе приведены все типы объектов схемы, при этом для каждого из них указаны общие и уникальные разновидности привилегий.

Таблица 5.10

Уникальные объектные привилегии

Объект схемы	Назначение	Уникальные привилегии	Системные (общие) привилегии
Каталоги	Предоставляет привилегии на выполнение операций над каталогами	READ, WRITE	–
Внешние таблицы	Предоставляет привилегии на выполнение операций над внешними таблицами	–	ALTER, SELECT
Библиотеки	Предоставляет привилегии на выполнение операций над библиотеками	–	EXECUTE

Окончание табл. 5.10

Объект схемы	Назначение	Уникальные привилегии	Системные (общие) привилегии
Материализованные представления	Предоставляет предварительно агрегированные сводные данные, участвующие в операциях бизнес-интеллекта	–	DELETE, FLASHBACK, INSERT, SELECT, UPDATE
Операторы	Предоставляет привилегии на выполнение операций над операторами	–	EXECUTE
Процедуры, функции и пакеты	Предоставляет привилегии на выполнение операций над тремя типами программных единиц: процедурами, функциями и пакетами	–	DEBUG, EXECUTE
Последовательности	Предоставляет привилегии на выполнение операций над последовательностями	–	ALTER, SELECT
Таблицы	Предоставляет привилегии на выполнение операций над таблицами	INDEX, ON COMMIT REFRESH, QUERY REWRITE	ALTER, DELETE, DEBUG, FLASHBACK, INSERT, REF_ ERENCES, SELECT, UPDATE
Пользовательские типы	Предоставляет привилегии на выполнение операций над пользовательскими типами	–	DEBUG, EXECUTE, UNDER
Представления	Предоставляет привилегии на выполнение операций над представлениями	–	DEBUG, DELETE, FLASHBACK, INSERT, REFERENCES, SELECT, UNDER

Привилегии и пользователи. Для назначения привилегий пользователю или роли применяется команда GRANT. Команда REVOKE позволяет лишить пользователя или роль привилегии.

Общие ключевые слова и инструкции. Ключевые слова и инструкции, которые могут применяться как в команде GRANT, так и в REVOKE:

– ALL PRIVILEGES – предоставляет или отбирает все системные привилегии, за исключением SELECT ANY DICTIONARY. Предоставляет все объектные привилегии, имеющиеся для данного объекта, ключевое слово PRIVILEGES является необязательным;

– *получатель* – один или несколько пользователей, одна или несколько ролей или ключевое слово PUBLIC, которое предоставляет или отбирает привилегии у всех пользователей БД. Если указывается несколько получателей, их следует разделить запятыми;

– *имя_столбца* – один или несколько столбцов, для которых предоставляется или отбирается привилегия доступа к объектам INSERT, REFERENCES или UPDATE. Если имя столбца не указано, то привилегия *предоставляется* на все столбцы таблицы или представления;

– *схема.объект* – указывает имя объекта, на который выдается или отбирается привилегия. Если схема не задана, то сервер *Oracle* считает, что объект находится в собственной схеме пользователя;

– DIRECTORY *имя_каталога* – указывает имя каталога, на который выдается или отбирается привилегия.

Инструкция GRANT предоставляет пользователю или роли привилегии или роли. Для того чтобы предоставить привилегию, пользователь должен предварительно получить привилегию или роль с указанием WITH ADMIN OPTION. Также можно предоставлять привилегии, если есть привилегия GRANT ANY PRIVILEGE (для системных привилегий), GRANT ANY ROLE (для ролей), GRANT OPTION (для объектов схемы) или вы являетесь владельцем объекта. Системные привилегии и привилегии доступа к объектам схемы не могут быть предоставлены в одной команде GRANT.

Синтаксис для предоставления системных привилегий или ролей будет таким:

GRANT {*системная_привилегия* | *роль* | ALL PRIVILEGES} TO *получатель*
[IDENTIFIED BY *пароль*] [WITH ADMIN OPTION];

Синтаксис для предоставления привилегий доступа к объектам схемы следующий:

GRANT {*объектная_привилегия* | ALL [PRIVILEGES]}
[*имя_столбца* [, *имя_столбца* …]]
ON {*схема.объект* | DIRECTORY *имя_каталога* |
JAVA [SOURCE | RESOURCE] [*схема.*]*объект*}
TO *получатель* [WITH GRANT OPTION] [WITH HIERARCHY OPTION];

Определение ключевых слов приведено в табл. 5.11.

Таблица 5.11

Ключевые слова инструкции GRANT

Ключевое слово	Назначение
IDENTIFIED BY	Может применяться для идентификации существующего пользователя по паролю или для создания нового пользователя с указанным паролем
ALL PRIVILEGES	Предоставляет пользователю или роли все привилегии, за исключением SELECT ANY DICTIONARY
WITH ADMIN OPTION	Позволяет пользователю выдавать или отбирать системную привилегию или роль, а также изменять и удалять роль
WITH GRANT OPTION	Позволяет получателю получить привилегии на все дочерние объекты указанного объекта схемы
SOURCE | RESOURCE	Разрешает доступ к исходным текстам *Java* или *Java*-ресурсу
WITH HIERARCHY OPTION	Позволяет пользователю получить привилегии на все дочерние объекты указанного в инструкции объекта схемы

Инструкция REVOKE аннулирует привилегии, ранее выданные пользователю или роли. Эта команда может аннулировать только те привилегии, которые были ранее выданы командой GRANT. Если отзывается роль у клиента, который использует ее в текущий момент, то роль остается, но уже не будет доступна клиенту после того, как он перестанет ей пользоваться.

Если несколько обладателей привилегии выдали ее пользователю, то для того чтобы привилегия стала недоступна пользователю, она должна быть аннулирована всеми выдавшими.

Для изъятия системных привилегий или ролей:

REVOKE {*системная_привилегия* | *роль* | ALL PRIVILEGES}
FROM *получатель*;

Для изъятия привилегий доступа к объектам:

REVOKE {*объектная_привилегия* | ALL [PRIVILEGES]}
[*имя_столбца* [,*имя_столбца* . . .]]
ON {*схема.объект* | DIRECTORY *имя_каталога* |
JAVA [SOURCE | RESOURCE] [*схема.*]*объект*}
FROM *получатель* [CASCADE CONSTRAINTS] [FORCE];

Определение ключевых слов приведено в табл. 5.12.

Таблица 5.12

Ключевые слова инструкции REVOKE

Ключевое слово	Назначение
ALL PRIVILEGES	Аннулирует все существующие системные привилегии для пользователя или роли
JAVA SOURCE | RESOURCE	Аннулирует доступ к исходным текстам *Java* и к *Java*-ресурсу
CASCADE CONSTRAINTS	Применяется только при отзыве привилегии REFERENCES или привилегий доступа к объектам ALL. Удаляет все ограничения, которые пользователь, лишаемый привилегий, определил для объекта
FORCE	Применяется при отзыве привилегии доступа к объектам EXECUTE для объектов пользовательских типов, имеющих зависимости от таблиц или типов. Приводит к тому, что все зависимые объекты помечаются как INVALID, запрещает доступ к данным зависимых таблиц, помечает все зависимые функциональные индексы как UNUSABLE

5.6. Роли в *Oracle*

Предоставление отдельных привилегий отдельным пользователям может существенно усложнить работу, особенно в корпоративных системах с большим количеством пользователей. Роли предназначены для того, чтобы упростить управление привилегиями.

Привилегии с помощью GRANT можно выдавать ролям, а затем также назначать роли пользователям. Ведение привилегий осуществляется на уровне ролей и затрагивает всех пользователей, которым назначены такие роли. Кроме того, в зависимости от контекста роли могут избирательно разрешаться и запрещаться для пользователей. Иными словами, посредством ролей можно объединять наборы привилегий и предоставлять их как единое целое. Например, у вас может существовать роль ADMIN, наделяющая администратора соответствующими полномочиями.

Роль может выдаваться другой роли. Если вы назначаете пользователю родительскую роль, он по умолчанию получает и все роли, выданные родительской роли. Пользователю можно назначить несколько ролей. Количество одновременно выдаваемых ролей ограничено параметром инициализации MAX_ENABLED_ROLES. Несколько ролей дают возможность одному пользователю в разные моменты времени применять разные наборы привилегий. Если роли выданы другие роли, то применение родительской роли подразумевает применение всех дочерних.

Команда ALTER USER позволяет задать одну или несколько ролей по умолчанию. Роли, назначенные по умолчанию, действуют, когда пользователь регистрируется в БД *Oracle*.

Системные роли. СУБД *Oracle* предлагает ряд предопределенных системных ролей, указанных в табл. 5.13.

Таблица 5.13

Предопределенные системные роли

Роль	Назначение
CONNECT	Включает в себя системные привилегии ALTER SESSION, CREATE CLUSTER, CREATE DATABASE LINK, CREATE SEQUENCE, CREATE SESSION, CREATE SYNONYM, CREATE TABLE и CREATE VIEW
RESOURCE	Включает в себя системные привилегии CREATE CLUSTER, CREATE INDEXTYPE, CREATE OPERATOR, CREATE PROCEDURE, CREATE SEQUENCE, CREATE TABLE и CREATE TYPE
DBA	Включает в себя все системные привилегии с указанием WITH ADMIN OPTION
CREATE TYPE	Включает в себя привилегии CREATE TYPE, EXECUTE, EXECUTE ANY TYPE, ADMIN OPTION и GRANT OPTION
EXP_FULL_DATABASE	Предназначена для предоставления всех привилегий, необходимых для выполнения полного и инкрементного экспорта БД
IMP_FULL_DATABASE	Предназначена для предоставления всех привилегий, необходимых для выполнения полного импорта БД
DELETE_CATALOG_ ROLE	Содержит в себе привилегию DELETE для таблицы аудита системы (AUD$)
SELECT_CATALOG_ ROLE	Включает в себя привилегию SELECT для таблицы аудита системы (AUD$)
RECOVERY_CATALOG_ OWNER	Предназначена для предоставления всех привилегий, необходимых владельцу каталога восстановления
HS_ADMIN_ROLE	Предназначена для защиты пакетов и представлений словарей данных гетерогенных сервисов

Определение ролей. Команды *SQL* позволяют создавать, изменять и удалять роли. Команда ALTER USER позволяет определить одну или несколько ролей пользователя по умолчанию. Роли, назначенные по умолчанию, действуют, когда пользователь регистрируется в БД *Oracle*.

Роль создается следующим образом:

CREATE ROLE *имя_роли* {NOT IDENTIFIED | IDENTIFIED
{EXTERNALLY | GLOBALLY | BY *пароль* | USING [*схема*.] *пакет*}};

Определение ключевых слов приведено в табл. 5.14.

Таблица 5.14

Ключевые слова инструкции CREATE ROLE

Ключевое слово	Назначение
NOT IDENTIFIED	Указывает, что для роли не требуется пароль
IDENTIFIED	Определяет, каким способом пользователь будет аутентифицирован, прежде чем ему будет разрешено активировать роль или назначить ее ролью по умолчанию
BY *пароль*	Локальный пользователь должен предоставить пароль для активирования роли
USING (*схема*.)*пакет*	Проверку пользователя осуществит пакет. Применяется для ролей приложений
EXTERNALLY	Внешний пользователь авторизуется сторонней службой, такой как операционная система
GLOBALLY	Пользователь авторизуется службой каталогов предприятия

Изменение определенной роли возможно инструкцией ALTER ROLE, которая изменяет способ аутентификации пользователя роли. Ключевые слова совпадают с ключевыми словами, приведенными для команды CREATE ROLE.

ALTER ROLE *имя_роли* {NOT IDENTIFIED | IDENTIFIED
{EXTERNALLY | GLOBALLY | BY *пароль* | USING [*схема*.] *пакет*}};

DROP ROLE удаляет роль из БД. Если роль используется в данный момент, то ничего не произойдет, но заново использовать роль пользователь уже не сможет.

DROP ROLE *имя_роли*;

SET ROLE разрешает пользователю применение одной или нескольких ролей или запрещает применение всех ролей.

SET ROLE {*роль* [IDENTIFIED BY *пароль*
[, *роль* [IDENTIFIED BY PASSWORD …]] |
ALL [EXCEPT *роль*[, *роль* …]] NONE};

где ALL – разрешает применение всех ролей, предоставленных пользователю;

EXCEPT – применяется для исключения некоторых ролей из списка разрешенных при помощи ключевого слова ALL;

NONE – запрещает применение всех ролей пользователя.

5.7. Аудит и другие способы обеспечения безопасности в *Oracle*

Аудит применяется как способ отслеживания использования БД и предупреждения о возможных проблемах с безопасностью.

В *Oracle* разрешен аудит четырех различных типов:

– Аудит команд, направленных к БД отдельными пользователями или всеми пользователями.

– Аудит привилегий – аудит использования системных привилегий отдельными пользователями или всеми пользователями.

– Аудит объектов схемы – аудит определенного набора команд *SQL* для конкретного объекта схемы.

– Детальный аудит (*fine-grained auditing*). Он появился в *Oracle*9*i*. Как и детальный контроль доступа, детальный аудит реализуется за счет определения предиката, вводящего ограничение для команд *SQL*, которые будут подвергнуты аудиту. Проводя такую политику аудита, можно сконцентрироваться на отслеживании действий с небольшим объемом жизненно важных данных. Уменьшение общего количества записей аудита облегчает выявление потенциальных проблем с безопасностью.

Для всех типов аудита *Oracle* вносит записи в журнал аудита БД, или таблицу *SYS.FGA_LOG*$, или в файл ОС в двоичном формате. Записи журнала аудита содержат различную информацию в зависимости от типа аудита и установленных параметров выполнения аудита.

Вне зависимости, включен ли аудит для БД, следующие действия всегда формируют записи в журнале аудита ОС:

1) запуск экземпляра сервера БД;
2) остановка экземпляра сервера БД;
3) доступ пользователей с привилегиями администратора.

Включать и выключать аудит можно при помощи команд AUDIT и NOAUDIT.

AUDIT *инструкция для команд SQL* | *инструкция для объектов схемы*
[BY SESSION | ACCESS] [WHENEVER [NOT] SUCCESSFUL];

где *инструкция_для_команд SQL* применяется для включения аудита команд или системных привилегий и имеет следующий формат:

{[*параметр команды* | ALL] [, …]} |
{[*системная_привилегия* | ALL PRIVILEGES] [, …]}
BY {*прокси_сервер* (, *прокси_сервер* …] ON BEHALF OF [{*пользователь* [, *пользователь* …]} | {*пользователь* [, *пользователь* …]}

инструкция_для_объектов_схемы применяется для включения аудита объектов схемы и имеет следующий формат:

{*параметр_объекта* [, *параметр_объекта* …] | ALL}
ON {*схема*.[*объект*] | DIRECTORY *имя_каталога* | DEFAULT}

Определение ключевых слов приведено в табл. 5.15.

Таблица 5.15

Ключевые слова инструкции AUDIT

Ключевое слово	Назначение
BY SESSION | ACCESS	Определяет, должны ли данные аудита записываться один раз за сеанс или при каждой попытке определенного типа доступа. Аудит для всех команд и всех привилегий для команд *DDL* может задаваться только как BY ACCESS
WHENEVER [NOT] SUCCESSFUL	Указывает, следует ли проверять только успешные или неудавшиеся команды *SQL*. Единственные неудавшиеся команды, отслеживаемые при помощи ключевого слова NOT, – это команды, которые не удается выполнить или приводят к возникновению ошибок из-за недостаточных привилегий или отсутствия объекта, на который приведена ссылка. По умолчанию проверяются все команды вне зависимости от того, выполнились они успешно или не удались по упомянутым причинам
BY *пользователь*	Задает аудит по одному или нескольким именам пользователя
BY *прокси_сервер* ON BEHALF OF	Задает аудит действий, выполняемых прокси-сервером от имени пользователя
параметр_команды	Включает аудит отдельных команд *SQL*

Синтаксис команды отмены аудита приведен ниже:

NOAUDIT *инструкция_для_команд_SQL* | *инструкция_для_объектов_схемы* [BY SESSION | ACCESS]

Описанные ниже возможности обеспечения безопасности по большей части относятся к вопросам администрирования БД *Oracle* или к специалистам, отвечающим за безопасность. Здесь приводится лишь краткий обзор таких функций, подробную информацию можно найти в документации *Oracle*.

Представления и хранимые процедуры. Самым распространенным способом ограничения доступа к данным в зависимости от их значений является использование представлений и хранимых процедур.

С помощью представлений определяется подмножество данных таблиц и предоставляется пользователю доступ только к представлениям. Также можно ограничить доступ к таблице, используя хранимую процедуру или пакет, выдав пользователям привилегии на них. Код хранимой процедуры может содержать собственный набор правил подтверждения правильности.

Детальный контроль доступа и политика безопасности. *Детальный контроль доступа* (*fine-grained access control*) предлагает способ контекстного обеспечения безопасности, который может быть реализован в коде приложения или в представлениях. Благодаря тому что детальный контроль доступа реализуется на уровне БД, он единообразно действует для всех приложений.

Детальный контроль доступа появился в версии *Oracle*8*i* и реализуется специальными правилами для таблиц, регламентирующими права доступа к этим таблицам.

Политика безопасности (*security policy*) реализуется программным модулем, который может предоставлять доступ на основе логического условия любого вида. Для всех команд *SQL*, выполняемых на БД, создается предикат (условие, добавляемое в инструкцию WHERE и ограничивающее доступ к данным), который автоматически может применяться для обеспечения безопасности на основе содержимого таблицы.

Виртуальные частные БД. Механизм контекстных мер безопасности позволяет создавать виртуальные частные БД (*VPD, Virtual Private Database*). Благодаря применению *VPD* сразу несколько пользователей могут видеть свои собственные представления одной или нескольких таблиц БД. Ранее говорилось о том, что подобные меры безопасности можно обеспечить за счет создания и поддержания представлений, но применение *VPD* избавляет от забот по созданию представлений и разграничению доступа к ним для отдельных пользователей и групп пользователей.

Метки безопасности и управление ими. Метки безопасности появились в *Oracle*9*i*. Они представляют собой расширение *VPD*; различие в том, что программные модули для поддержки *VPD* уже написаны и действуют для значений единственного столбца, содержащего метки. Поэтому для реализации меток безопасности не требуется специального программирования.

Метки безопасности относятся к дополнительным компонентам *Oracle*9*i Enterprise Edition. Policy Manager*, инструмент администрирования с графическим интерфейсом пользователя, входящий в состав *Enterprise Manager*, также появился в версии *Oracle*9*i*. Он реализует управление метками безопасности.

Контекст приложения (*application context*). Введен в *Oracle*8*i*. Позволяет установить для пользователя некоторые атрибуты, которые будут действовать в течение всего пользовательского сеанса. Применяя такие атрибуты для предоставления доступа, можно создать роль для конкретного приложения, которая продолжает существовать в БД на всем протяжении работы приложения. Контекст приложения может использоваться для реализации детального контроля доступа.

Инструментальное средство *LogMiner*. Позволяет применять команды *SQL* для анализа событий в журнале БД, т. е. позволяет отслеживать транзакции по мере их обработки или выявлять, какие конкретно функции вызвали изменение данных. Утилита *LogMiner* появилась в *Oracle*8*i*. С помощью *LogMiner* (наряду с журналами аудита) можно определить, что происходило в БД. В версии *Oracle*9*i* *LogMiner* был включен в *Enterprise Manager*.

Продукт *Oracle Advanced Security*, который ранее назывался *Secure Network Services*, а затем *Advanced Network Services,* – это пакет расширения, предлагающий мощные средства шифрования данных. *Oracle Advanced Security* обеспечивает дополнительные возможности по обеспечению безопасности в трех областях:

– Сетевая безопасность. Шифрование сообщений, передаваемых службами *Oracle Net Services*, реализация *SSL*_шифрования (*Secure Sockets Layer* – протокола защищенных сокетов) и поддержка *RADIUS*, *Kerberos*, смарт-карт, карточек-идентификаторов (*tokencards*) и биометрической аутентификации.

– Безопасность пользователей предприятия. Включает применение разнообразных сторонних средств поддержки каталогов, таких как *LDAP*_каталоги, с помощью которых можно реализовать возможность однократной регистрации. *Oracle Advanced Security* включает в себя сервис каталогов *Oracle Internet Directory* (*OID*).

– Безопасность инфраструктуры открытых ключей. Включает поддержку стандартных сертификатов *X*.509 версии 3. *Oracle* работает с основными поставщиками сервисов инфраструктуры открытых ключей (*Public Key Infrastructure – PKI*), такими как *Baltimore Technologies* и *VeriSign*, для обеспечения координации с их доверенными корневыми сертификатами.

Oracle Advanced Security внедряет эти сервисы на уровне *Oracle Net Services*, который реализует взаимодействие между сервером и клиентом. Кроме того, *Oracle Advanced Security* можно использовать с драйвером *Thin JDBC,* не содержащим *Oracle Net Services*.

Oracle Advanced Security включает в себя *Oracle Enterprise Security Manager*, графический интерфейс пользователя для управления доменами и пользователями предприятия.

Контрольные вопросы и задания к разделу 5

1. Какие принципы защиты данных предлагает стандартный *SQL*?
2. Чем различаются по возможностям команды создания и изменения пользователя?
3. Запишите команду создания пользователя *test* с паролем 123 и требованием изменить его для доступа к БД?
4. Какие инструкции используются для работы с привилегиями?
5. Как называется и осуществляется процесс распознавания авторизованных пользователей в СУБД?
6. Какие системные пользователи создаются при установке СУБД *Oracle*?
7. Запишите инструкцию ALTER TABLE, которая изменит пароль пользователя *test* и назначит ему по умолчанию роль *test_role*.
8. Для чего используются профили?
9. Перечислите параметра пароля, которые могут быть настроены в профиле.
10. Какими командами можно назначить профиль пользователю?

11. Какие виды привилегий выделяют в *Oracle*?

12. Что дает использование ключевых слов SYSDBA и SYSOPER?

13. Для чего применяются объектные привилегии FLASHBACK и REFERENCES?

14. Какие ключевые слова в инструкции GRANT позволяют пользователю выдавать или отбирать системные привилегии у других пользователей?

15. Запишите команду предоставления всех системных привилегий пользователю *test* с правом их передачи.

16. Для чего и где используются ключевые слова WITH HIERARCHY OPTION?

17. Приведите последовательность действий при необходимости создания роли и назначения ее пользователю.

18. Запишите команду, запрещающую пользователю использование всех ролей.

19. Какие виды аудита бывают?

20. Что всегда автоматически записывается в журнал аудита БД?

6. ОБЩИЕ СВЕДЕНИЯ И СТРУКТУРА ЯЗЫКА *PL/SQL*

6.1. Преимущества языка

Язык *PL/SQL* (*Procedural Language extensions to the Structured Query Language* – процедурное расширение структурированного языка запросов) – это хорошо структурированный, удобный и доступный язык, тесно интегрированный с БД *Oracle*. Он стал первым языком, получившим поддержку в СУБД *Oracle*, и до сих пор остается наиболее популярным средством создания программных модулей, таких как процедуры, функции, пакеты и триггеры.

Преимущества использования языка *PL/SQL* заключается в следующем:

1) позволяет работать с переменными и типами данных, в том числе с коллекциями, записями, массивами, курсорами, исключительными ситуациями, параметрами;

2) добавляет управляющие структуры ветвления, циклов, переходов; позволяет организовать циклы по записям, обрабатывая каждую по отдельности;

3) позволяет создавать процедуры, функции, триггеры, пакеты;

4) дает возможность создавать объектные типы и методы;

5) позволяет защитить программы, используя шифрование, хранить код программы на сервере;

6) экономит сетевой трафик, так как имеет блочную (пакетную) структуру;

7) в целом улучшает производительность, группируя запросы и вычислительные блоки в единую конструкцию, которая будет размещаться и выполняться на сервере.

Приложения, написанные на *PL/SQL*, полностью переносимы. Язык обеспечивает высокий уровень безопасности, доступ к предопределенным пакетам *SQL*, поддержку объектно-ориентированного программирования и разработку веб-приложений и серверных страниц.

В качестве среды разработки может использоваться *SQL*Plus* – программа-интерпретатор командной строки, в которой могут выполняться команды *SQL* и *PL/SQL* в интерактивном виде или из сценария. Также существует *iSQL*Plus* – веб-утилита, позволяющая работать через графический интерфейс в любом браузере, в том числе при использовании облачной версии сервера СУБД *Oracle*.

Более профессиональной средой разработки является *Oracle SQL Developer* – интегрированная среда разработки на языках *SQL* и *PL/SQL* с возможностью администрирования БД. Корпорация *Oracle* предоставляет продукт бесплатно. Сама среда написана на языке программирования *Java*.

Для демонстрации работы языка *PL/SQL* будут использоваться таблицы схемы HR (*Human Resources* — «человеческие ресурсы»*),* которая представлена на рис. 6.1. Код создания и заполнения этой схемы можно скачать в [7], также она легко устанавливается в облачной версии *Oracle XE*, как *Sample Dataset* через пункт меню *Utilities* раздела *SQL Workshop*.

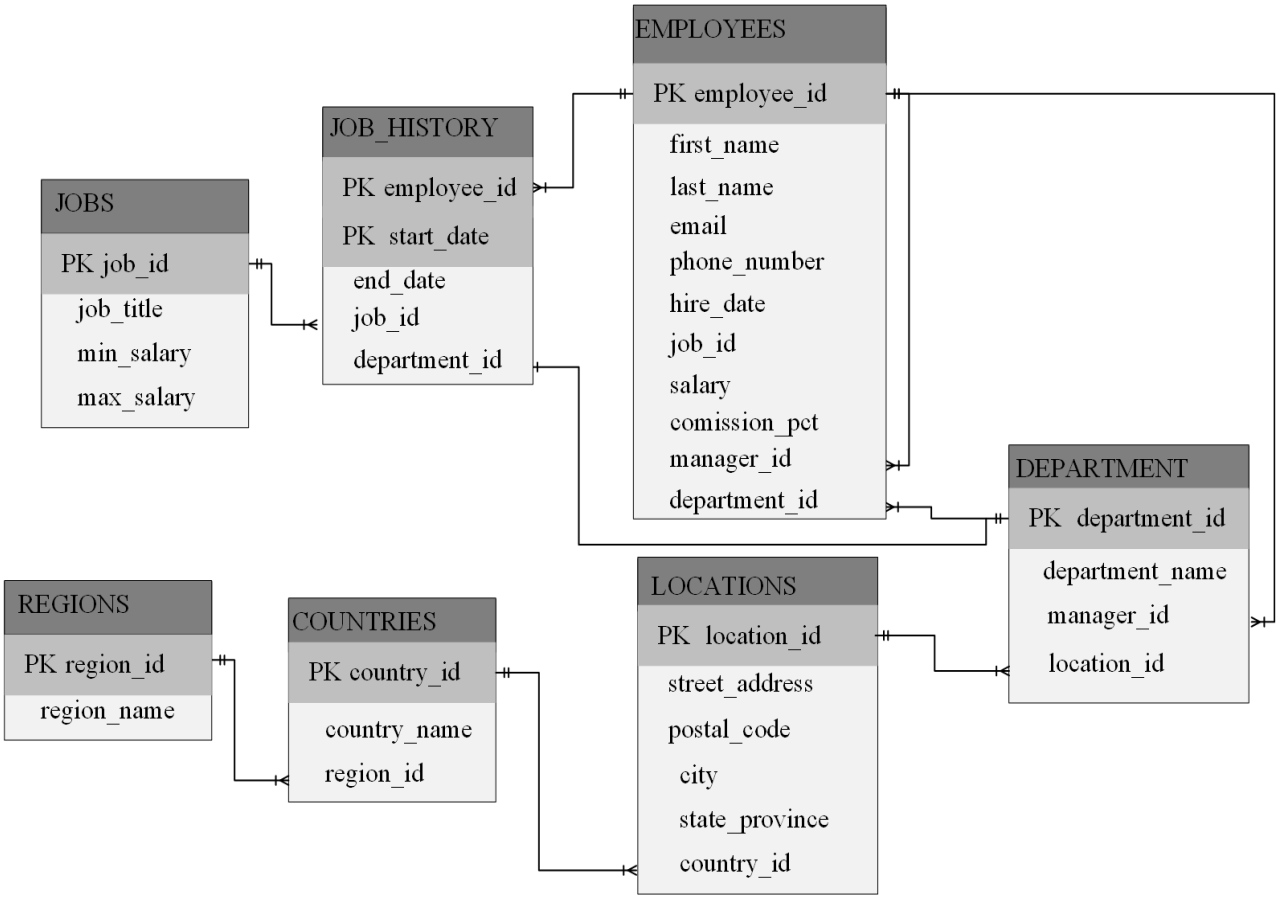


Рис. 6.1. Схема данных HR

6.2. Лексические единицы

Лексическая единица – это последовательность символов, входящих в набор символов, разрешенных в *PL/SQL*.

В языке *PL/SQL* используются следующие буквы, цифры, специальные символы и символы-разделители:

– буквы A–Z, a–z;

– цифры 0–9;

– специальные символы: - ! @ # $ % & * () _ – + = | [] {} : ; " ' < > , . ? / ^;

– пробельные символы: пробел, табуляция, возврат каретки, новая строка.

Все лексические единицы делятся на четыре группы:

1) идентификаторы;

2) литералы;

3) разделители;

4) комментарии.

1. Идентификаторы – это имена объектов *PL/SQL*, включая константы, скалярные переменные, составные переменные (записи или коллекции), исключения, процедуры, функции, пакеты, типы, курсоры, зарезервированные слова и метки. Идентификаторы обладают следующими характеристиками:

– могут состоять не более чем из 30 символов;

– не могут содержать пробельные символы (пробел, табуляцию, возврат каретки, новую строку);

– должны начинаться с буквы;

– могут содержать знаки: доллара ($), подчеркивания (_) или диеза (#);

– не могут быть зарезервированными (ключевыми) словами;

– не чувствительны к регистру.

Если идентификаторы заключены в двойные кавычки, то в силе остается только первое из всех этих правил. Другими словами, в закавыченный идентификатор может входить любой печатный символ, за исключением самих двойных кавычек.

В целом настоятельно рекомендуется не использовать без крайней необходимости такие нестандартные идентификаторы, поскольку это влечет появление дополнительных проблем. Так, например, применение кавычек будет оправдано, если какой-нибудь столбец в БД называется зарезервированным словом *PL/SQL* и его нельзя переименовать или создать представление, в котором столбец будет переименован.

2. Литералы – это конкретные символьные, числовые или логические значения, которые не являются идентификаторами. Существует несколько различных типов литералов:

– числовые;

– строковые;

– логические;

– литералы интервалов дат и времени.

Литералы могут иметь неопределенное значение NULL. Не существует литералов для составных типов, так как они существует только во внутреннем представлении.

Числовой литерал – это целое или действительное значение, которое может быть присвоено без преобразования переменной с типом NUMBER. Эти литералы используются в арифметических выражениях. В целых литералах может использоваться знак «+» или «–» и цифры. В действительных литералах дополнительно используются десятичная точка или знак экспоненциальной формы – «e», после которого можно указывать только целый литерал.

Символьный (строковый) литерал состоит из одного или нескольких символов, заключенных в одинарные кавычки ('Rick', 'Это интересная книга'). В отличие от идентификаторов, литералы чувствительны к регистру. Для того чтобы поместить одинарную кавычку в строковый литерал, надо расположить две одинарные кавычки друг за другом ('Mikes''string').

Перечень правил, которыми следует руководствоваться при включении одинарной кавычки в литерал:

– чтобы включить одинарную кавычку в литерал, используют две подряд идущие одинарные кавычки;

– если нужно включить одинарную кавычку в начало или конец литерала, используют три подряд идущие одинарные кавычки;

– для создания строкового литерала, состоящего из одной одинарной кавычки, используют подряд четыре одинарные кавычки;

– чтобы создать литерал, состоящий из двух одинарных кавычек, используют подряд шесть одинарных кавычек.

Две одинарные кавычки подряд – это не то же самое, что символ двойной кавычки, который в строковом литерале не имеет специального значения, а интерпретируется так же, как буква или цифра.

Кроме этого, *Oracle* позволяет пользователю самостоятельно определять символ, который будет использоваться в качестве кавычек с помощью оператора q'. Выглядеть такое переопределение («экранирование») может, например, так: q'[литерал содержит кавычку' и имеет длину > 0]'.

Особенности применения:

– можно использовать как строчную букву q, так и заглавную (Q);

– после q должна идти обычная одинарная кавычка. Следующая одинарная кавычка определяет завершение области действия оператора q;

– сразу после одинарной кавычки должен идти символ, выбранный пользователем для применения в качестве кавычек. Этот символ может быть любым, за исключением пробела, перевода строки или табуляции (в том числе и одинарная кавычка). Если пользователь выбрал символ «(», «{», «[» или «<», то закрываться кавычки должны соответствующим символом «)», «}», «]» и «>». В других случаях закрытие кавычек производится при помощи того же символа, который использовался для их открытия.

Строковые литералы могут быть присвоены переменным с типом CHAR или VARCHAR2 без преобразования.

Логические литералы TRUE, FALSE, NULL могут быть присвоены только логическим (булевым) переменным, обычно используются с управляющими операторами типа IF, LOOP.

В *Oracle*9*i* были введены два новых интервальных типа данных – это INTERVAL YEAR TO MONTH и INTERVAL DAY TO SECOND. Они представляют временные интервалы, выраженные в годах и месяцах либо в днях, часах, минутах, секундах и долях секунды. Литералы этого типа имеют форму INTERVAL YEAR (*точность_года*) TO MONTH или INTERVAL DAY (*точность дня*) TO SECOND (*точность долей секунды*).

3. Разделители (ограничители) – это символы или последовательности символов, имеющие специальное значение в *PL/SQL*: отделяют друг от друга идентификаторы. Имеющиеся в *PL/SQL* разделители перечислены в табл. 6.1

Таблица 6.1

Разделители в *PL/SQL*

Разделитель	Описание
;	Признак конца оператора, ограничитель для команд и объявлений
+ – * / ** =	Арифметические операторы

Окончание табл. 6.1

Разделитель	Описание
<> != ^= ~=	Операторы неравенства
< > <= >= <>	Операторы сравнения
||	Операторы конкатенации
:=	Оператор присваивания
()	Ограничитель начала и конца выражения или списка
,	Разделитель элементов в списках
'	Ограничитель литерала
“	Ограничитель «закавыченного» идентификатора
<< >>	Ограничители метки
:	Ограничитель переменной привязки (связи)
%	Ограничитель атрибута
.	Признак компонента, в частности поля записи или элемента пакета
@	Признак связи БД
=>	Признак связывания параметров при вызове (ассоциация)
..	Оператор диапазона в цикле FOR

4. Комментарии – тип документирования программ и отдельных операторов с целью облегчения понимания их работы. Они могут быть записаны в одной строке или занимать несколько строк. Однострочные и многострочные комментарии, представленные в табл. 6.2, отделяются от остальной программы ограничителями и эффективно игнорируются *PL/SQL* при компиляции.

Таблица 6.2

Комментарии

Комментарий	Описание
--	Признак однострочного комментария
/* */	Ограничители многострочного комментария

6.3. Структура блока

Каждая программа на *PL/SQL* представляет собой блок, состоящий из стандартного набора элементов, обозначенных ключевыми словами. Блок определяет область видимости объявленных переменных, порядок обработки и передачи исключений.

Структура блока определяется следующей синтаксической конструкцией:

[CREATE OR REPLACE *имя* [(*параметр тип_данных* [, *параметр тип_данных* …]) {IS | AS}] -- раздел заголовка

[[DECLARE]
переменная тип_переменной;
[*переменная тип_переменной*; …]] -- раздел объявлений
BEGIN
выполняемый_код;
[*выполняемый_код*; ...] -- раздел выполнения
[EXCEPTION
код_исключения;
[*код_исключения*; ...]] -- раздел исключений
END;

Назначение секций:

– раздел заголовка определяет имя блока, необходим для именованных блоков и недопустим в анонимных;

– раздел объявлений содержит объявления переменных, констант, курсоров, типов и локальных программ, используемых в данном блоке. Может отсутствовать;

– раздел выполнения содержит выполняемый код. Обычно присутствует, но необязателен в спецификациях пакетов и типов;

– раздел исключений определяет обработку исключений. Может отсутствовать.

Разделы выполнения и исключений могут содержать в себе подблоки (вложенные блоки).

Самый простой блок, который возможно создать, будет выглядеть так:

```
BEGIN
NULL;
END;
/
```

Необходимо обратить внимание, что все представленные в пособии примеры будут выполняться как в командной строке *SQL*Plus*, так в *iSQL*Plus* (включая версию в облаке), с той лишь разницей, что последняя строка выполняемого блока в виде наклонной черты **обязательна только в *SQL*Plus*** и далее в примерах будет отсутствовать!

6.4. Раздел заголовка

В секции заголовка указывается имя блока *PL/SQL* или программного модуля. Наличие или отсутствие этой секции определяется типом блока *PL/SQL*.

Есть два основных типа блоков.

1. Анонимный блок создается динамически и выполняется только один раз. Как правило, такой блок создается в клиентской программе для вызова под-

программ, хранящихся в БД, и не может быть вызван из-за пределов содержащего его блока. Необязательная секция объявлений начинается ключевым словом DECLARE. Пусть анонимный блок имеет вид

```
DECLARE
Today DATE DEFAULT SYSDATE;
BEGIN
-- Вывести дату
DBMS_OUTPUT.PUT_LINE ('Сегодня  '|| today);
END;
```

Важно отметить, что для корректной работы такого блока в *SQL*Plus* необходимо один раз в сессию включать серверный вывод результатов пакета DBMS_OUTPUT работы командой

```
SET SERVEROUTPUT ON
```

2. Именованный блок может быть разделен на три подвида:

– помеченные блоки – это анонимные блоки с меткой. Они создаются обычно динамически и выполняются только один раз. Метка позволяет ссылаться на переменные блока, которые иначе были бы недоступны. Метка задается перед ключевым словом DECLARE или после ключевого слова END;

– подпрограммы – это процедуры и функции, которые могут храниться в БД как автономные объекты, как часть пакета или как метод объектного типа. Подпрограммы обычно не изменяются и выполняются неоднократно, могут быть объявлены в других блоках. Независимо от того, где они объявлены, выполняются явно посредством вызова процедуры или функции, для обозначения секции объявлений в них не применяется ключевое слово DECLARE. Пример блока этого типа:

```
CREATE OR REPLACE PROCEDURE show_the_date IS
Today DATE DEFAULT SYSDATE;
BEGIN
-- вывести дату
DBMS_OUTPUT.PUT_LINE ('сегодня '||today);
END show_the_date;
```

Вызвать на выполнение в *SQL*Plus* можно, например, так:

```
EXEC show_the_date;
```

Команда EXEC в таком виде будет работать только в *SQL*Plus*, но, по сути, это запуск хранимой процедуры с помощью анонимного блока, т. е.

```
BEGIN
show_the_date;
END;
```

– триггеры – именованные блоки, которые ассоциируются с некоторым событием, происходящим в БД. Обычно не изменяются и выполняются многократно неявным образом при наступлении соответствующих событий (INSERT, UPDATE, DELETE, CREATE, ALTER, DROP, запуск и остановка сеанса работы сервера БД и др.).

Параметры. В секции заголовка именованного блока могут быть объявлены параметры. Если они присутствуют, то заключены в скобки и разделены запятыми.

В процедурах, функциях и курсорах параметры служат для передачи информации между подпрограммой и вызывающей программой. Параметры перечисляются в списке, содержащем один или более параметров. В определение каждого параметра вводят его имя, тип данных, назначение и (необязательно) значение по умолчанию.

Синтаксис определения параметров:

имя_параметра [*назначение*] [NOCOPY] *тип_данных* [(:= | DEFAULT | *значение*)]

Ключевые слова:

Назначение – указывает, будет ли параметр использован для получения или возврата значений. Может принимать следующие значения:

– IN – входной параметр только для чтения. Значение фактического параметра можно использовать внутри программы, но оно не может быть изменено. Этот режим установлен по умолчанию;

– OUT – выходной параметр, внутри подпрограммы рассматривается как неинициализированная переменная, т. е. можно как записать в него значение, так и считать из него;

– IN OUT – параметр для чтения и записи. Значение фактического параметра передается в процедуру при ее вызове. Внутри процедуры формальный параметр рассматривается как инициализированная переменная, при чем можно как записать в него значение, так и считать значение из него.

Если в ходе выполнения процедуры или функции возникают исключения, то исключения, присвоенные параметрам типа OUT и IN OUT, теряются, если только не дана подсказка NOCOPY.

Ключевое слово NOCOPY – это подсказка компилятору использовать передачу параметра по ссылке, а не по значению. По умолчанию *PL/SQL* передает параметр типа IN OUT по значению, создавая его копию в подпрограмме. Когда параметр содержит много данных для передачи, например является коллекцией или объектом, то такое копирование замедляет работу, требует много памяти. NOCOPY указывает *PL/SQL*, что параметр передается по ссылке при помощи указателя на единственный экземпляр параметра. Недостаток применения NOCOPY в том, что при возникновении исключения в процессе выполнения программы, которая изменяет параметр типа OUT или IN OUT значения реальных параметров не откатываются, так как они передавались по ссылке, а не по значению.

тип_данных – может быть любым типом данных *PL/SQL* или пользовательским типом, может быть ограничен в размере. Фактический размер параметра определяется вызывающей программой или ограничивается атрибутом %TYPE.

Пример написания именованного блока с параметрами:

```
CREATE OF REPLACE PROCEDURE empid_to_name
(in_id              employee.employee_id%TYPE; -- компилируется успешно
, out_last_name     VARCHAR2;       -- компилируется успешно.
, out_first_name    VARCHAR2(10); -- не компилируется.
) IS
…
```

Длина параметров out_last_name и out_first_name будет определяться вызывающей программой, например, в анонимном блоке:

```
DECLARE
surname         VARCHAR2(10);
first_name      VARCHAR2(10);
BEGIN
Empid_to_name(10, surname, first_name);
END;
```

Ключевое слово DEFAULT указывает значение по умолчанию для входного (IN) параметра. Если входной параметр имеет значение по умолчанию, то можно не указывать его значение явно при вызове программы. Если при вызове значение не указано, автоматически будет поставлено значение по умолчанию.

Следует различать формальные и фактические параметры. Формальные параметры – это имена, объявляемые в заголовке процедуры или функции, а фактические – это значения или выражения, помещаемые в список параметров при вызове функции или процедуры. Рассмотрим примеры использования различных типов параметров.

1. Пример использования параметра IN:

```
CREATE OR REPLACE FUNCTION testInParameter (arg IN INT)
RETURN INT IS
ret INT;
BEGIN
  DBMS_OUTPUT.PUT_LINE (arg);  -- нормально
  ret := 1;
  ret := ret + arg;          -- нормально
  arg := arg + 1;            -- ошибка
  RETURN ret;
END testInParameter;
```

Просмотреть ошибки компиляции в *SQL*Plus* можно командой SHOW ERRORS или выполнив запрос к представлению user_errors (этот вариант работает и в *iSQL*Plus*):

```
SELECT * FROM user_errors;
```

2. Пример использования параметра OUT:

```
CREATE OR REPLACE PROCEDURE testOutParameter (arg OUT INT) IS
BEGIN
  DBMS_OUTPUT.PUT_LINE (arg);  -- ничего не выведет, arg = NULL
  arg := arg + 1;
  DBMS_OUTPUT.PUT_LINE (arg);  -- arg все еще NULL
  arg := 10;
  DBMS_OUTPUT.PUT_LINE (arg);  -- выводит 10
END testOutParameter;
```

Выполним процедуру с помощью анонимного блока:

```
DECLARE
  res INT;
BEGIN
  res := 5;
  DBMS_OUTPUT.PUT_LINE (res);
  testOutParameter (res);
  DBMS_OUTPUT.PUT_LINE (res);
END;
```

3. Пример использования параметра IN OUT:

```
CREATE OR REPLACE PROCEDURE testInOutParameter (arg IN OUT INT)
IS
BEGIN
DBMS_OUTPUT.PUT_LINE (arg);  -- выведет переданное в процедуру значение
  arg := 10;                 -- нормально
  arg := arg + 5;            -- нормально
  DBMS_OUTPUT.PUT_LINE (arg);  -- выведет присвоенное значение
END;
```

Выполним вызов в анонимном блоке:

```
DECLARE
  res INT;
BEGIN
  res := 5;
  testInOutParameter (res);
  DBMS_OUTPUT.PUT_LINE (res);
END;
```

6.5. Секция объявлений

В секцию помещаются объявления всех переменных, констант, курсоров и локальных подпрограмм, используемых в данном блоке. Наличие секции объявлений необязательно: если в программе нет перечисленных выше компонент, то она может отсутствовать.

Объявление переменной обязательно содержит указания на тип данных. В *PL/SQL* в дополнении ко всем разрешенным в *SQL* типам применяются также несколько собственных типов. Необходимо помнить, что в большинстве случаев возможности и ограничения для типов *PL/SQL* и БД идентичны, но некоторые из них имеют настолько различные возможности хранения, что это может рано или поздно проявиться и навредить работе.

Объявление переменных. Прежде чем использовать переменную, ее необходимо объявить в секции объявлений программного блока или, как глобальную, в спецификации пакета. Встретив объявление переменной, *PL/SQL* выделяет область памяти для хранения значения и сопоставляет ее имя, обращаясь к которому, можно извлекать значение и изменять его. **В одном операторе может быть объявлена только одна переменная.** Для объявления переменной применяется следующий синтаксис:

имя_переменной тип_данных [CONSTANT] [NOT NULL]
[{:=| DEFAULT} *начальное_значение*]

Сразу после объявления переменной ей по умолчанию присваивается значение NULL. Инициализация всех переменных – характерная черта *PL/SQL*, этим он отличается от таких языков, как *C* и *Ada*. Инициализировать переменную значением отличным от NULL позволяет оператор присваивания или указание ключевого слова DEFAULT:

```
counter BINARY_INTEGER :=0;
priority VARCHAR2(8) DEFAULT 'LOW';
```

В объявление переменной можно добавить ограничение NOT NULL, означающее, что значение NULL для этой переменной недопустимо. Если такое ограничение указано, то необходимо явно присвоить начальное значение этой переменной.

При объявлении констант (которое осуществляется добавлением ключевого слова CONSTANT) необходимо указать начальное значение, которое в дальнейшем не может быть изменено, например:

```
min_order_qty NUMBER (1) CONSTANT :=5;
```

Объявления с ограничениями и без ограничений. Указание на тип данных может содержать ограничение, а может не содержать. Ограничение заключается в указании допустимого размера, масштаба или точности, не превышающих соответствующие значения для ограниченного типа. Например:

```
total_sales      NUMBER(15,2);  -- Есть ограничение.
emp_id           VARCHAR2(9);   -- Есть ограничение.
company_number   NUMBER;        -- Нет ограничения.
book_title       VARCHAR2;      -- Ошибка.
```

Данные, объявленные с ограничениями, занимают меньше места, чем неограниченные. Не все типы данных могут быть указаны без ограничений. Например, нельзя объявить переменную с типом VARCHAR2 – необходимо указать ее максимальный размер.

Общепринятые соглашения об именованиях переменных *PL/SQL* приведены в табл. 6.3.

Таблица 6.3

Общепринятые соглашения об именованиях переменных *PL/SQL*

Префикс	Тип	Пример
*v_*	Стандартная переменная	*v_empno*
*c_*	Константа	*c_sysdate*
*p_*	Переменная *SQL*Plus*	*p_infield*
*g_*	Глобальная переменная *SQL*Plus*	*g_field_pos*

Закрепленные объявления типов (использование атрибутов). Для того чтобы закрепить (*anchor*) тип данных скалярной переменной (которая держит одно значение) за другой переменной или столбцом таблицы или представления БД, применяется атрибут %TYPE. Атрибут %ROWTYPE позволяет закреплять объявление за курсором или таблицей.

При объявлении переменной (но не в определении столбца) инструкция NOT NULL может следовать за закреплением типа, в этом случае необходимо, чтобы объявления типов содержали значения по умолчанию. Значение по умолчанию для закрепленного объявления может отличаться от значения по умолчанию основного объявления.

Области действия и области видимости переменных. Область действия (*scope*) переменной – это фрагмент программы, в котором возможно обращение к данной переменной. Для переменной *PL/SQL* это фрагмент с момента ее объявления до конца блока. Когда переменная выходит из своей области действия, *PL/SQL* освобождает память, используемую для хранения переменной, и ссылки на нее становятся невозможны. На рис. 6.2 показано, что областью действия переменной *v_Character* является только внутренний блок; после ключевого слова END внутреннего блока она становится недоступной. Область действия переменной *v_Number* распространяется до ключевого слова END внешнего блока. Внутренний блок является областью действия обеих переменных.

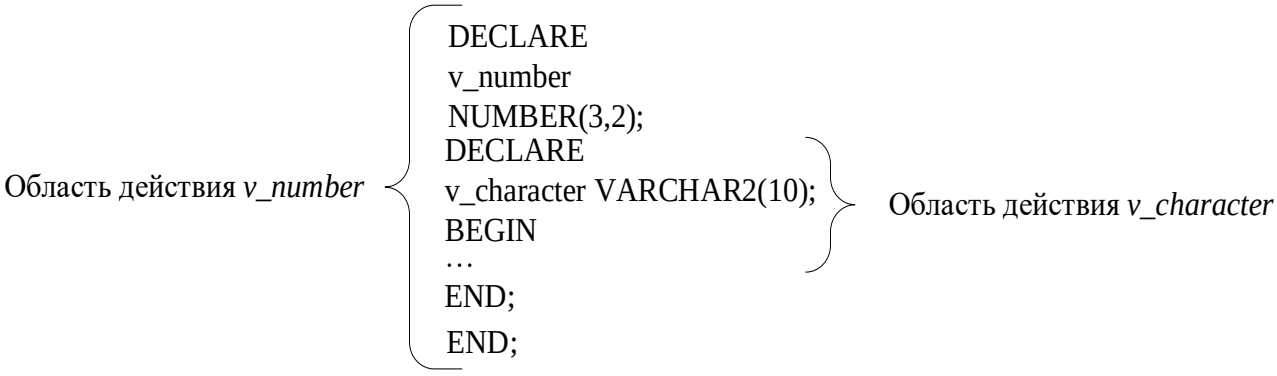


Рис. 6.2. Область действия переменных

Область видимости (*visibility*) переменной – это фрагмент программы, в котором возможно обращение к этой переменной без использования ее квалифицированного имени. Область видимости всегда лежит в области действия; если переменная находится вне области своего действия, она невидима. Рассмотрим рис. 6.3.

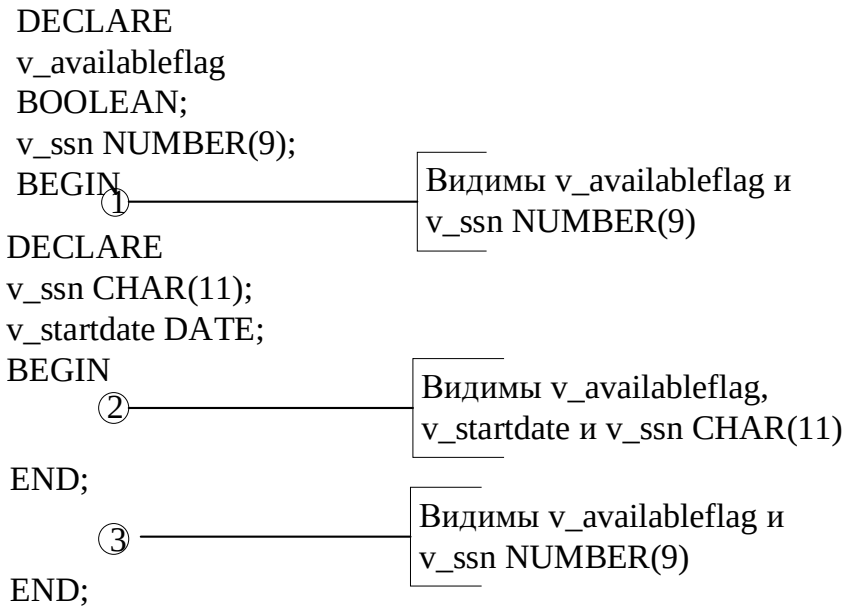


Рис. 6.3. Область видимости переменных

В точке 1 переменные *v_availableflag* и *v_ssn* находятся в границах своих областей действия и видимы. В точке 2 обе эти переменные находятся в пределах своих областей действия, но видима только переменная *v_availableflag*. При объявлении переменной *v_ssn* типа CHAR(11) объявление NUMBER(9) становится скрытым. В этой точке все четыре переменные находятся в границах своих областей действия, но видимы только три: *v_availableflag*, *v_startdate* и *v_ssn* CHAR(11). В точке 3 переменные *v_startdate* и *v_ssn* CHAR(11) находятся вне пределов своих областей действия и поэтому больше невидимы. Здесь действуют и видимы те же переменные, что и в точке 1: *v_availableflag* и *v_ssn* NUMBER(9). Чтобы обратиться в программе к переменной, которая находится в своей области

действия, но невидима, необходимо использовать ее квалифицированное (полное) имя. На рис. 6.4 приведен тот же блок, что и на рис. 6.3, но для внешнего блока создана метка «*1_outer*». В точке 2 переменная *v_ssn* NUMBER(9) невидима. Обратиться к ней можно с помощью метки следующим образом: *1_outer.v_ssn*.

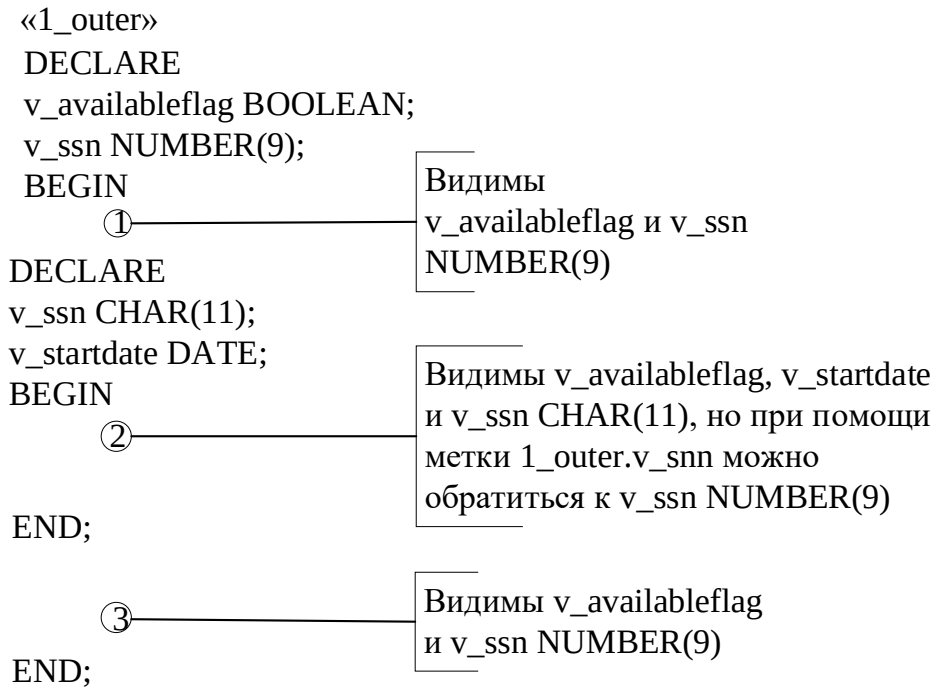


Рис. 6.4. Использование меток для обращения к переменным

Контрольные вопросы и задания к разделу 6

1. Какие возможности языка относятся к преимуществам *PL/SQL*?
2. Какие виды лексических единиц различают в *PL/SQL*?
3. Какие специальные символы используются в идентификаторах?
4. Какие разделы блока *PL/SQL* являются обязательными?
5. Какие бывают блоки *PL/SQL* в зависимости от раздела заголовка?
6. Как объявить символьную константу *den_ed* со значением «рубли»?
7. Что обозначает ограничитель /* */?
8. Запустить анонимный и именованный блоки для вывода текущей даты, изменить формат вывода таким образом, чтобы увидеть время.
9. Запустить примеры использования различных типов параметров.
10. Написать анонимный блок с меткой, аналогичный представленному на рис. 6.4, вывести переменные, находящиеся и в области видимости, и в области действия.

7. ТИПЫ ДАННЫХ *PL/SQL*

Все типы данных *PL/SQL* определены в пакете STANDARD, где также содержатся встроенные *SQL*-функции и функции преобразования типов.

Типы данных могут быть отнесены к одной из следующих категорий:

1) скалярные;
2) составные;
3) ссылочные;
4) большие объекты.

7.1. Скалярные типы

Скалярные типы. Скалярным типом называют тип, содержащий одно значение. Скалярные типы делятся на подтипы: символьные (строковые), числовые, логические (булевы), дата – время. Символьные типы *PL/SQL* приведены в табл. 7.1.

Таблица 7.1

Символьные типы

Название типа	Подтипы	Описание
CHAR[(*размер*)]	CHARACTER(*размер*)	Сохраняет строки символов фиксированной длины. Максимальный размер 32 767 байт, однако для БД ограничен 2000 байт
VARCHAR2(*размер*)	VARCHAR(*размер*), STRING	Сохраняет строки символов переменной длины. Максимальный размер 32 767 байт, однако в БД может хранить только 4000 байт
LONG	–	Хранит последовательность символов переменной длины, максимально 32 760 байт, не совпадает с типом LONG БД, размер которого больше
RAW(*размер*)	–	Хранит двоичные данные фиксированной длины и может содержать до 32 Кбайт данных. В БД содержит только 2 Кбайт
LONG RAW	–	Хранит до 32 Кбайт двоичных данных, не совпадает с типом LONG RAW БД, размер которого больше

Окончание табл. 7.1

Название типа	Подтипы	Описание
ROWID	–	Хранит значение ROWID записи БД без конвертирования его в символьный тип, поддерживает физические идентификаторы строк
UROWID	–	Поддерживает как физические, так и логические идентификаторы строк
NCHAR	–	Содержит национальные символьные данные фиксированной длины
NVARCHAR2	–	Содержит национальные символьные данные переменной длины

Типы данных RAW и LONG RAW используются для данных, которые не должны ни интерпретироваться *Oracle*, ни преобразовываться при передаче данных между различными системами. Эти типы данных предназначены для двоичных данных или однобайтных строк. Например, LONG RAW можно использовать для хранения графики, звука, документов или массивов двоичных данных; их интерпретация зависит от их использования.

Тип RAW эквивалентен VARCHAR2, а LONG RAW эквивалентен LONG, с тем исключением, что *SQL*Net* (который соединяет пользовательские сессии с инстансом) и утилиты экспорта и импорта не выполняют преобразований при передаче данных RAW или LONG RAW. Напротив, *SQL*Net* и утилиты импорта/экспорта автоматически преобразуют данные CHAR, VARCHAR2 и LONG из набора символов БД в набор символов сессии пользователя (установленным параметром NLS_LANGUAGE или командой ALTER SESSION) и обратно, если эти наборы символов различны.

Когда *Oracle* автоматически преобразует данные RAW или LONG RAW в тип данных CHAR или из него (как в случае, когда данные RAW вводятся как литерал в предложении INSERT), эти данные рассматриваются как шестнадцатеричные цифры, каждая из которых представляет полубайт (четыре бита). Например, 1 байт данных RAW с битовым представлением 1100 1011 вводится и отображается как 'CB'.

Данные LONG RAW не могут индексироваться, однако данные RAW можно индексировать.

Тип данных ROWID. Каждой строке некластеризованной таблицы в БД *Oracle* назначается уникальный ROWID, соответствующий физическому адресу данной строки. В случае кластеризованных таблиц строки разных таблиц, если они хранятся в одном и том же блоке данных, могут иметь одинаковый ROWID.

Каждая таблица в БД *Oracle* внутренне имеет псевдостолбец с именем ROWID; этот псевдостолбец не виден при выводе таблицы с помощью предложения SELECT * FROM ... или предложения DESCRIBE. Однако адрес каждой строки можно извлечь *SQL*-запросом, используя ключевое слово ROWID как имя столбца, например:

```
SELECT ROWID, last_name FROM employees
WHERE ROWNUM < 4;
```

В результате получим:

ROWID	LAST_NAME
AEyKPTAP8AAAADLAAD	Abel
AEyKPTAP8AAAADMAAU	Ande
AEyKPTAP8AAAADNAAe	Atkinson

Псевдостолбец ROWID использует двоичное представление физического адреса для каждой выбираемой строки, но, как видно из примера, при запросах это двоичное представление преобразуется в шестнадцатеричное представление VARCHAR2.

Для хранения данных ROWID требуется 80 бит (10 байт). Идентификаторы ROWID состоят из четырех компонентов: номера объекта (32 бита), относительного номера файла (10 бит), номера блока (22 бита) и номера строки (16 бит). Эти идентификаторы отображаются как 18-символьные последовательности, указывающие местонахождение данных в БД, причем каждый символ представлен в формате *base*-64, состоящем из букв *A*–*Z*, *a*–*z*, цифр 0–9, символов «+» и «/». Первые шесть символов – это номер объекта данных, следующие три – относительный номер файла, следующие шесть – номер блока, последние три – номер строки. Чтобы увидеть эту информацию, можно воспользоваться возможностями встроенного пакета DBMS_ROWID следующим образом:

```
SELECT ROWID,
    DBMS_ROWID.ROWID_OBJECT (ROWID) "Объект",
    DBMS_ROWID.ROWID_RELATIVE_FNO (ROWID) "Файл",
    DBMS_ROWID.ROWID_BLOCK_NUMBER (ROWID) "Блок",
    DBMS_ROWID.ROWID_ROW_NUMBER (ROWID) "Строка"
 FROM employees;
```

При каждом перемещении, экспорте, импорте строки, а также при выполнении любых других операций, которые приводят к изменению ее местонахождения, изменяется ROWID строки, поскольку она занимает другое физическое положение.

Нельзя устанавливать значение псевдостолбца ROWID в предложениях INSERT или UPDATE. Значения ROWID в псевдостолбце ROWID внутренне используются *Oracle* в разнообразных операциях, хотя к значениям псевдостолбца ROWID можно обращаться как к другим столбцам таблицы (в списках SELECT и фразах WHERE), эти значения не хранятся физически в БД и не являются данными БД.

Числовые типы *PL/SQL* приведены в табл. 7.2.

Таблица 7.2

Числовые типы

Название типа	Подтипы	Описание
BINARY_INTEGER (–2147483647.. 2147483647)	–	Сохраняет целые числа со знаком, используется для значений, которые не будут храниться в БД, а только применяются в вычислениях. Использует библиотечную арифметику
NUMBER (*p*, *s*) (*p* – точность, *s* – масштаб)	DEC, DECIMAL, DOUBLE PRECISION, FLOAT(*точность*), INTEGER, INT, NUMERIC, REAL, SMALLINT	*p*, *s* либо не указываются, либо указываются вместе. Сохраняет числа с фиксированной и плавающей точкой, имеют перечисленные подтипы. Использует библиотечную арифметику, хранятся в десятичном формате. Если применяются в вычислениях, то автоматически преобразуются в BINARY_INTEGER и обратно
PLS_INTEGER (–2147483647.. 2147483647)	NATURAL, NATURALN, POSITIVE, POSITIVEN, SIGNTYPE	Сохраняет целые числа со знаком, имеет тот же формат и диапазон, что и BINARY_INTEGER, однако если при выполнении некоторой операции переполнить значение этого типа, то возникнет ошибка. Для ускорения вычислений использует машинную арифметику. NATURAL и NATURALN сохраняют только неотрицательные целые числа, причем последний запрещает NULL-значения. POSITIVE и POSITIVEN сохраняют только положительные целые числа, причем последний запрещает NULL-значения. SIGNTYPE сохраняет только –1, 0, 1

Окончание табл. 7.2

Название типа	Подтипы	Описание
BINARY_DOUBLE	–	Тип двойной точности с плавающей точкой, используется для научных вычислений
BINARY_FLOAT	–	Тип одинарной точности с плавающей точкой, используется для научных вычислений

К логическим (булевым) типам относится единственный тип BOOLEAN, который принимает значения TRUE, FALSE, NULL.

К числу подтипов дата – время относятся типы DATE, TIMESTAMP, INTERVAL, которые практически совпадают с соответствующими типами БД.

7.2. Составные типы

Составные типы отличаются от скалярных типов тем, что у них имеются внутренние компоненты – атрибуты. К составным типам относятся записи, вложенные таблицы, индексные таблицы и массивы переменной длины.

Записи. Запись (*record*) – это составная структура данных, которая и по сути, и по структуре похожа на строку таблицы БД. Она состоит из нескольких элементов, называемых полями (*fields*). Запись в целом не имеет значения – значение имеет каждое отдельное поле, а запись предоставляет возможность хранения этих значений и доступ к ним как к группе. Для того чтобы использовать запись, необходимо определить ее и объявить переменную такого типа.

Существуют три типа записей:

– на основе таблицы;

– на основе курсора;

– определяемые программистом.

Записи на основе таблицы и курсора не надо определять явно, так как они неявно определяются с той же структурой, что таблица или курсор. Переменные этих типов объявляются при помощи атрибута %ROWTYPE. Поля записей соответствуют столбцам таблицы или столбцам списка выборки для курсора.

```
DECLARE
v_jobsrecord jobs%ROWTYPE;
```

Записи, определяемые программистом, должны быть явно объявлены командой TYPE в секции объявлений *PL/SQL* или в спецификации пакета. Затем можно объявить переменные такого типа или использовать его как часть другого типа:

```
DECLARE
TYPE name_rectype IS RECORD
(prefix VARCHAR2(15),
fname VARCHAR2(15),
lname VARCHAR2(30),
suffix VARCHAR2(10),
hire_date DATE);

new_emp_rec name_rectype;
```

Также можно создавать вложенные записи – это записи, состоящие из полей, которые представляют собой записи. Вложение записей – мощное средство нормализации структур данных и сокрытия сложности в программах *PL/SQL*.

Инициализация записей. На отдельные поля записи можно ссылаться посредством точечной нотации:

имя записи. *имя поля*

Отдельные поля записи могут быть прочитаны и записаны. Они могут стоять как слева, так и справа от оператора присваивания. Запись целиком может быть присвоена другой записи того же типа, но сравнение двух записей при помощи логического оператора невозможно. Например, такое присваивание разрешено:

```
new_emp_rec_1 := new_emp_rec;
```

Но приведенное ниже сравнение уже не разрешено (вместо этого следует сравнивать отдельные поля записей):

```
IF new_emp_rec_1 = new_emp_rec
THEN
....
END IF;
```

Присваивание значений записям и полям записей может выполняться четырьмя способами:

1) применить оператор присваивания для присваивания значений полю:

```
new_emp_rec.hire_date := SYSDATE;
```

2) выполнить SELECT INTO для целой записи или отдельных полей:

```
SELECT  first_name, last_name, hire_date
INTO new_emp_rec.fname, new_emp_rec.lname, new_emp_rec.hire_date
FROM emp;
```

3) выполнить FETCH INTO для целой записи или отдельных полей:

FETCH emp_cur INTO new_emp_rec;

4) присвоить все поля одной переменной записи другой переменной записи того же типа.

Коллекции *PL/SQL*. Коллекция (*collection*) – это составная структура данных, которая ведет себя как список или одномерный массив (*PL/SQL* не поддерживает традиционные массивы).

В *PL/SQL* существует три типа коллекций: ассоциативные (индексные) массивы (*associative arrays*), вложенные таблицы (*nested tables*) и массивы переменной размерности (*variable arrays*).

Коллекции реализуются как типы (TYPE). Как и для других типов, определяемых пользователем, необходимо сначала определить тип, а затем объявить экземпляр такого типа. Определение типа может быть сохранено в БД или помещено в программу *PL/SQL*. Каждый экземпляр такого типа является коллекцией. Для лучшего понимания сравнение типов коллекции приведено в табл. 7.3 [6].

Таблица 7.3

Сравнение типов коллекций

Характеристика	Ассоциативный массив	Вложенная таблица	Массив переменной размерности
1	2	3	4
Размерность	Одноразмерный	Одноразмерный	Одноразмерный
Использование в SQL	Нет	Да	Да
Использование как тип данных поля в таблице	Нет	Да, данные сохраняются в отдельной таблице	Да, данные сохраняются в той же таблице
Неинициализированное состояние	Пустой (не NULL); элементы не определены	NULL; обращение к элементам невозможно	NULL; обращение к элементам невозможно
Инициализация	Автоматически при объявлении	Через конструктор, оператор FETCH и назначение	Через конструктор, оператор FETCH и назначение
В *PL/SQL* элементы ссылаются через	BINARY_INTEGER, PLS_INTEGER, VARCHAR2	Положительные целые между 1 и 2 147 483 647	Положительные целые между 1 и 2 147 483 647
Разрывы в данных	Да	Изначально нет; после удаления да	Нет

Окончание табл. 7.3

1	2	3	4
Ограниченный своими размерами	Нет	Может быть расширен	Да
Может переназначить значение любому элементу в любое время	Да	Нет, возможно сначала необходимо выполнить EXTEND	Нет, возможно сначала необходимо выполнить EXTEND; не может выйти за свои внешние границы, заданные при объявлении
Каким образом расширяется	Назначается значение элементу с новым индексом	Используется встроенная процедура EXTEND (или TRIM для уменьшения), нет предопределенных пределов	EXTEND (или TRIM), но только до максимального размера, заданного при объявлении
Сравнение на равенство	Нет	Нет	Нет
Сохраняет порядок после сохранения и получения из БД	Недоступно	Нет	Да

Ассоциативный массив – это одномерная неограниченная разреженная коллекция однородных элементов, которая доступна только в программах *PL/SQL*, но не в БД. Этот массив больше похож на списки и карты распределения в классических языках программирования, чем на обычные массивы. Ассоциативный массив может индексироваться при помощи типов BINARY_INTEGER, PLS_INTEGER или VARCHAR2. Он является ключом к использованию оператора FORALL или фразы BULK COLLECT, которые разрешают групповую пересылку данных из БД в программный модуль.

Полный синтаксис:

CREATE [OR REPLACE] TYPE *имя_типа* IS TABLE OF *тип_элемента* [NOT NULL] INDEX BY [BINARY_INTEGER| PLS_INTEGER | VARCHAR2 (*размер*)];

Ключевые слова:

имя_типа – любой разрешенный идентификатор, который впоследствии будет использоваться для объявления коллекции;

тип_элемента – это тип элементов коллекции. Все элементы должны быть одного типа; обычно бывает скалярный тип данных, объектный тип или ссылочный объектный тип. Если элементы представляют собой объекты, то сам объектный тип не может иметь в качестве атрибута коллекцию. К явно запрещенным типам данных элементов коллекций относятся BOOLEAN, NCHAR, NCLOB, NVARCHAR2, BCURSOR, TABLE и VARRAY;

NOT NULL – означает, что в коллекции данного типа не может быть элементов со значением NULL. Однако она может быть атомарно пустой (неинициализированной).

Вложенная таблица – это одномерная неограниченная коллекция однородных элементов, доступная как в *PL/SQL*, так и в БД в виде столбца или таблицы. Вложенные таблицы изначально являются плотными (имеют последовательные индексы), но могут стать разреженными в результате удалений. Самыми близкими соответствиями со стандартными языками программирования для них являются множества и множества с повторяющимися элементами.

[CREATE [OR REPLACE]] TYPE *имя_типа* IS TABLE OF *тип_элемента* [NOT NULL];

Значения ключевых слов аналогичны ассоциативному массиву.

Массив переменной размерности (VARRAY) – одномерная ограниченная коллекция однородных элементов, которая доступна как в *PL/SQL*, так и в БД. Массивы VARRAY всегда ограничены и никогда не бывают разреженными. В отличие от вложенных таблиц порядок их элементов сохраняется при сохранении их в БД и извлечении оттуда. Массив переменной длины является массивом в традиционном смысле языков программирования. Для ссылки на элементы структуры в них используются последовательные значения индексов:

[CREATE [OR REPLACE]] TYPE *имя_типа* IS {VARRAY|
VARYING ARRAY} (*максимально_элементов*) OF *тип_элемента* [NOT NULL];

Ключевые слова:

максимально_элементов – это положительное целое число, которое задает максимальное количество элементов массива переменной длины и является обязательным для указания.

7.3. Работа с коллекциями

Создание коллекций. После создания типа любой из представленных выше коллекций необходимо создать экземпляр коллекции, объявив переменную, например:

```
TYPE index_table IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
indtab intex_table;
```

где *indtab* – это экземпляр типа *index_table*.

При помощи атрибута %TYPE коллекции можно связать с ней тип данных другой коллекции, например:

```
TYPE varray1 IS VARRAY (20) OF NUMBER;
v1 varray1;
v2 v1% TYPE;
```

Если определение *v1* изменится, то вместе с ним изменится и определение *v2*.

Функции и методы коллекций. Действия с коллекциями можно выполнить посредством следующих функций:

– CAST – преобразует тип одной коллекции в тип другой коллекции;

– MULTISET – отображает таблицу БД на коллекцию. Применяя MULTISET и CAST, можно извлекать строки из таблицы БД в виде столбца с типом коллекции. Можно применять MULTISET и CAST для извлечения вложенных столбцов и использования их в триггерной логике;

– TABLE – отображает коллекцию на таблицу БД; обратная функция к MULTISET.

Функция TABLE позволяет использовать мощный инструмент *SQL*-запросов применительно к коллекциям, что, в отличие от последовательной обработки, значительно удобнее.

В следующем примере показано, как можно использовать функции CAST, MULTISET для выборки данных из таблиц в коллекции. Использование коллекции сокращает количество обращений к БД. Особенно если выдаются не только отчеты, но и выполняется многократное обновление информации.

Сначала создадим объект и вложенную таблицу:

```
/*объект для хранения данных о сотруднике*/
CREATE OR REPLACE TYPE inf_type AS OBJECT (
 first_name   VARCHAR2 (20),    dater DATE);

/*вложенная таблица для хранения информации о сотрудниках */
CREATE OR REPLACE TYPE lst_inf_type AS TABLE OF inf_type

DECLARE
  lst_empl   lst_inf_type := lst_inf_type ();
  n          INTEGER;
BEGIN
  -- увеличили размер буфера вывода (только в SQL*Plus)
  DBMS_OUTPUT.ENABLE(100000);
  SELECT CAST (MULTISET (SELECT   first_name, hire_date
  FROM employees ORDER BY first_name) AS lst_inf_type)
   INTO lst_empl FROM dual;

  DBMS_OUTPUT.PUT_LINE ('Список сотрудников: ');
```

```
        FOR i IN lst_empl.FIRST .. lst_empl.LAST
        LOOP
          DBMS_OUTPUT.put_line (   i
                                || ':'
                                || RPAD (lst_empl (i).first_name, 20, ' ')
                                || ':'
                                || TO_CHAR (lst_empl (i).dater, 'dd.mm.yyyy'));
        END LOOP;

        /* получим список сотрудников, проработавших более 5 лет*/
        n    := 0;
        DBMS_OUTPUT.PUT_LINE ('Список сотрудников, проработавших 5 и более лет: ');

        FOR cd IN (SELECT   first_name, TO_CHAR (lc.dater, 'dd.mm.yyyy') dater
          FROM TABLE (lst_empl) lc
          WHERE TRUNC (MONTHS_BETWEEN (SYSDATE, dater) / 12) >= 5
          ORDER BY lc.dater)
        LOOP
          n := n + 1;
          DBMS_OUTPUT.PUT_LINE (   n
                                || '.'
                                || RPAD (cd.first_name, 20, ' ')
                                || ':'
                                || cd.dater);
        END LOOP;

        DBMS_OUTPUT.PUT_LINE ('Список сотрудников, принятых в текущем году');
        n    := 0;
        FOR cd IN (SELECT   first_name, TO_CHAR (dater, 'dd.mm.yyyy') dater
                     FROM TABLE (lst_empl)
                    WHERE EXTRACT (YEAR FROM SYSDATE) = EXTRACT (YEAR FROM dater) ORDER BY dater)
        LOOP
          n    := n + 1;
          DBMS_OUTPUT.PUT_LINE (   n
                                || '.'
                                || RPAD (cd.first_name, 20, ' ')
                                || ':'
                                || cd.dater);
        END LOOP;
      END;
```

Также можно осуществлять выборку данных из коллекции (функция TABLE).

```
CREATE OR REPLACE TYPE a_data IS TABLE OF VARCHAR2(30);

DECLARE
        v_data a_data := a_data ('Dmitry', 'Sergei', 'Yulya' );
BEGIN
-- column_value – это имя поля, возвращаемого
-- TABLE (CAST (collection))
        FOR c1 IN (
SELECT column_value name FROM TABLE (CAST( v_data AS a_data)) )
        LOOP
        DBMS_OUTPUT.PUT_LINE ('Looking for values: ' || c1.name);
        END LOOP;
END;
```

Коллекции также поддерживают ряд методов, приведенных в табл. 7.4.

Таблица7.4

Методы коллекций

Метод	Описание
EXISTS(*x*)	Возвращает TRUE, если существует *x*-й элемент вложенной таблицы или изменяемого массива. В противном случае возвращает FALSE
COUNT	Возвращает число элементов, находящихся в данный момент во вложенной таблице или в изменяемом массиве
LIMIT	Возвращает максимальное число элементов, которое может содержать изменяемый массив
FIRST	Возвращает первый компонент вложенной таблицы или изменяемого массива
LAST	Возвращает последний компонент вложенной таблицы или изменяемого массива
PRIOR(*x*)	Возвращает компонент, предшествующий *x*-му компоненту вложенной таблицы или изменяемого массива
NEXT(*x*)	Возвращает компонент, следующий за *x*-м компонентом вложенной таблицы или изменяемого массива
EXTEND(*x*, *y*)	Добавляет *x* копий *y*-го элемента вложенной таблицы или изменяемого массива
TRIM(*x*)	«Вырезает» *x* элементов с конца вложенной таблицы или изменяемого массива
DELETE(*x*)	Удаляет некоторые или все элементы вложенной таблицы или изменяемого массива

Для вызова этих методов применяется синтаксис точечной нотации:

имя коллекции.имя метода[(*параметры*)]

Добавление и удаление элементов. Можно добавить элементы в ассоциативный массив, просто сославшись на новые индексы. Для добавления элементов во вложенные таблицы или массивы VARRAY необходимо сначала увеличить коллекцию с помощью функции EXTEND, а затем присвоить новому элементу значение посредством одного из ранее представленных методов.

Для удаления элемента вложенной таблицы, независимо от его позиции, применяется функция DELETE. Удалять элементы позволяет и функция TRIM, но только для удаления из конца коллекции.

Не стоит использовать для одной коллекции DELETE и TRIM – результаты могут быть совсем не теми, которые ожидались.

Использование некоторых методов с коллекциями рассмотрим на следующем примере:

```
DECLARE
  lst_child   lst_inf_type := lst_inf_type ();
BEGIN
  DBMS_OUTPUT.PUT_LINE ('count in lst_child =' || lst_child.COUNT);
  lst_child.EXTEND;
  lst_child (lst_child.LAST)    :=
            inf_type ('Анна', TO_DATE ('08.09.2023', 'dd.mm.yyyy'));
  lst_child.EXTEND;
  lst_child (lst_child.LAST)    :=
          inf_type ('Николай', TO_DATE ('05.06.2016', 'dd.mm.yyyy'));
  lst_child.EXTEND;
  lst_child (lst_child.LAST)    :=
            inf_type ('Ольга', TO_DATE ('28.11.2000', 'dd.mm.yyyy'));
  DBMS_OUTPUT.PUT_LINE ('Список детей ');
  FOR i IN lst_child.FIRST .. lst_child.LAST
  LOOP
    DBMS_OUTPUT.PUT_LINE (   i
                          || ':'
                          || lst_child (i).first_name
                          || ':'
                          || TO_CHAR (lst_child (i).dater, 'dd.mm.yyyy'));
  END LOOP;

  DBMS_OUTPUT.PUT_LINE ('Дети до 5 лет :');

  FOR cd IN (SELECT   first_name, TO_CHAR (dater, 'dd.mm.yyyy') dater
   FROM TABLE (lst_child)
   WHERE TO_CHAR (dater, 'yyyy') >= TO_CHAR (SYSDATE, 'yyyy')-5
          ORDER BY dater)
```

```
  LOOP
    DBMS_OUTPUT.PUT_LINE (cd.first_name || ':' || cd.dater);
  END LOOP;
END;
```

Вложенные коллекции. Вложенные коллекции – это коллекции, содержащиеся в элементах, которые являются членами другой коллекции. Они представляют собой мощное средство реализации объектно-ориентированных программных конструкций в программах *PL/SQL*.

Приведем простой пример определения вложенной коллекции:

```
CREATE TYPE books IS TABLE OF VARCHAR2(64);
CREATE TYPE our_books IS TABLE OF books;
```

Инициализация коллекций. Объявление ассоциативного массива означает также и его инициализацию. Инициализация вложенной таблицы и массива VARRAY может быть выполнена следующим образом:

– явно, при помощи конструктора;

– неявно, посредством извлечения из БД;

– неявно, посредством прямого присваивания другой переменной коллекции.

Конструктор – это встроенная функция, имя которой совпадает с именем коллекции. Она создает коллекцию из переданных ей элементов. Например, можно создать вложенную таблицу регионов и явно инициализировать ее тремя элементами при помощи конструктора:

```
DECLARE
TYPE regions_tab_t IS TABLE OF VARCHAR2(30);
regions_tab_t ('AMERICA', 'RUSSIA', 'POLAND');
BEGIN...
```

Применяя второй способ инициализации, можно создать вложенную таблицу регионов и неявно инициализировать ее при помощи извлечения данных из БД:

```
--Создать хранение в БД по типу вложенной таблицы.
CREATE TYPE regions_tab_t  IS TABLE OF VARCHAR2(32);

--Инициализировать коллекцию регионов данными из таблицы regions.
DECLARE
basic_regions   regions_tab_t;
BEGIN
SELECT region_name INTO basic_regions FROM regions;
END;
```

Третий метод инициализации – инициализируем таблицу неявно, используя присваивание из существующей коллекции:

```
DECLARE
basic_regions regions_tab_t :=
regions_tab_t ('AMERICA', 'RUSSIA', 'POLAND');

my_regions regions_tab_t;

BEGIN
my_regions := basic_regions;
my_regions(1) := 'CHINA';
END;
```

7.4. Ссылочные типы и типы больших объектов

Ссылочные типы данных REF CURSOR и REF. Для хранения скалярных и составных типов переменных выделяется определенная область памяти. Переменная присваивает ей имя, по которому происходит ссылка на переменную. При этом нельзя отменить выделение памяти и одновременно сохранить возможность работы с переменной. Ссылочные типы обходят этот запрет, т. е. переменные, объявленные как ссылочные, могут указывать на различные области памяти. Ссылочный тип – это то же самое, что и указатель в языке программирования *C*. С помощью такого типа можно именовать области хранения данных во время выполнения программы.

Переменные типа REF CURSOR называются курсорными переменными. Можно определить курсорную переменную как принадлежащую типу SYS_REFCURSOR и получать с ее помощью различные наборы данных из процедуры или функции. Например:

```
CREATE OR REPLACE PROCEDURE employees_sel
(cv_results IN OUT SYS_REFCURSOR) IS
BEGIN
OPEN cv_results FOR
SELECT employee_id, first_name, last_name FROM employees;
END;
```

Тип REF используется с объектными типами. Значение REF следует представлять как указатель на экземпляр объекта в объектной таблице или объектном представлении.

Большие объекты. *Oracle* поддерживает следующие четыре типа для больших объектов:

– BFILE – для внешнего двоичного файла, сохраняет указатели на объекты LOB, управляемые файловыми системами, внешними по отношению к СУБД;

– BLOB – для внутреннего двоичного объекта;

– CLOB – для внутреннего символьного объекта;

– NCLOB – для внутреннего символьного объекта, учитывающего национальный набор символов.

Типы LOB от типа LONG отличаются главным образом тем, что при выборе значения любого LOB-типа посредством оператора SELECT возвращается указатель, а не само значение; кроме того, типы LOB могут быть и внешними.

Любой объект LOB состоит из двух частей: данных и указателя на эти данные, называемого локатором.

Типы BLOB, CLOB или NCLOB могут использоваться как для столбца БД, так и для переменной *PL/SQL*.

Для загрузки объекта LOB предусмотрен пакет *PL/SQL* DBMS_LOB.

Процедуры и функции для работы с LOB-типами приведены в табл. 7.5.

Таблица 7.5

Процедуры и функции для работы с LOB-типами

Синтаксис	Описание
APPEND (*d*1, *d*2)	Добавляет *d*2 к *d*1
COMPARE (*d*1, *d*2, *n*, *pos*1, *pos*2)	Сравнивает *n* байт значений *d*1 и *d*2
COPY (*d*, *s*, *n*, *dp*, *sp*)	Копирует *n* байт из *d* в *s*.
FILEOPEN (*bdata*, *m*)	Открывает объект типа BFILE в режиме, указанном параметром *m*
LOADFROMFILE (*bdata*1, *data*2, *n*, *pos*1, *pos*2)	Копирует *n* байт объекта типа BFILE *bdata*1 в любой объект LOB *data*2
GETLENGTH (*data*)	Возвращает длину указанного объекта LOB
READ (*data*, *n*, *pos*, *buf*)	Читает из объекта *data n* байт
WRITE (*data*, *n*, *pos*, *buf*)	Копирует из буфера *buf n* байт
EMPTY_CLOB (), EMPTY_BLOB ()	Создают «пустой» объект указанного типа

Кроме всех рассмотренных типов в *PL/SQL* для написания объектно-ориентированных программ широко применяются **объектные типы**, которые обеспечивают уровень абстракции для лежащих в их основе структур данных. По структуре объектный тип похож на пакет и имеет спецификацию и тело. Объектный тип может иметь **атрибуты** и **методы**. Атрибуты определяют характеристики объекта, а методы действуют на структуры данных объектного типа и обеспечивают возможность всех взаимодействий между приложением и данными объекта, физически они являются процедурами или функциями, описывающими действия или поведение объекта. Таким образом, объектным типом называется определение или шаблон объекта, а не сам объект. В зависимости от того, как они были созданы, объектные типы могут быть временными (*transient*) или постоянными (*persistent*). Временные объектные типы доступны только во время выполнения создавшей их программы, а постоянные сохраняются в БД постоянно (существуют объектные таблицы, столбцы, представления). Более подробную информацию о создании объектных типов можно найти в источнике [11].

7.5. Преобразование типов данных

Компилятор *PL/SQL* может выполнять преобразования между различными семействами скалярных типов данных. В пределах одного семейства типы данных можно преобразовывать произвольно, но с учетом ограничений, налагаемых на переменные. Так, переменную типа CHAR(10) нельзя преобразовать в переменную типа VARCHAR2(1), так как не будет хватать места для хранения ее значения. Ограничения точности и масштаба также могут препятствовать преобразованию, например, переменной NUMBER(3, 2) в переменную NUMBER(3) и наоборот. В случае нарушения ограничения компилятор *PL/SQL* не выдаст сообщения об ошибке, однако ошибки могут появиться во время выполнения программы – все зависит от значений преобразуемых переменных.

Существует два способа преобразования типов данных: явный и неявный.

Явное преобразование типов данных. Встроенные функции преобразования, доступные в *SQL*, доступны и в *PL/SQL*. При необходимости их можно применять для явного преобразования типов данных, относящихся к различным семействам типов. Более подробную информацию о функциях преобразования и примеры их использования можно найти в табл. 7.6 [6].

Таблица 7.6

Функции преобразования типов данных *PL/SQL* и *SQL*

Функция	Выполняемое преобразование
CHARTOROWID	Строку в значение типа ROWID
CONVERT	Строку из одного набора символов в другой
FROM_TZ	Значение типа TIMESTAMP добавляет информацию о часовом поясе, преобразуя его тем самым в значение типа TIMESTAMP WITH TIME ZONE
HEXTORAW	Значение из шестнадцатеричного формата в значение типа RAW
MULTISET	Таблицу БД в коллекцию
NUMTODSINTERVAL	Число (или числовое выражение) в литерал INTERVAL DAY TO SECOND
NUMTOYMINTERVAL	Число (или числовое выражение) в литерал INTERVAL YEAR TO MONTH
RAWTOHEX, RAWTONHEX	Значение типа RAW в шестнадцатеричный формат
REFTOHEX	Значение типа REF в символьную строку, содержащую его шестнадцатеричное представление
ROWIDTOCHAR, ROWIDTONCHAR	Двоичное значение типа ROWID в символьную строку
TO_CHAR, TO_NCHAR (*числовая версия*)	Число в строку (VARCHAR2 или NVARCHAR2 соответственно)

Окончание табл. 7.6

Функция	Выполняемое преобразование
TO_CHAR, TO_NCHAR (*версия для дат*)	Дату в строку
TO_CHAR, TO_NCHAR (*символьная версия*)	Данные из набора символов БД в набор символов национального языка
TO_BLOB	Значение типа RAW в значение типа BLOB
TO_CLOB, TO_NCLOB	Значение типа VARCHAR2, NVARCHAR2 или NCLOB в значение типа CLOB (либо NCLOB)
TO_DATE	Строку в дату
TO_DSINTERVAL	Значение типа CHAR, VARCHAR2, NCHAR или NVARCHAR2 в значение типа INTERVAL DAY TO SECOND
TO_LOB	Значение типа LONG в LOB
TO_MULTI_BYTE	Однобайтные символы исходной строки в их многобайтные эквиваленты (если это возможно)
TO_NUMBER	Строку в число
TO_RAW	Значение типа BLOB в значение типа RAW
TO_SINGLE_BYTE	Многобайтные символы исходной строки в соответствующие однобайтные символы
TO_TIMESTAMP	Символьную строку в значение типа TIMESTAMP
TO_TIMESTAMP_TZ	Символьную строку в значение типа TIMESTAMP_TZ
TO_YMINTERVAL	Значение типа CHAR, VARCHAR2, NCHAR или NVARCHAR2 в значение типа INTERVAL YEAR TO MONTH

Неявное преобразование типов данных. В *PL/SQL* осуществляется автоматическое преобразование типов данных разных семейств, когда это возможно. Например, при помощи блока, приведенного ниже, извлекается значение заработной платы сотрудника с номером 101:

```
DECLARE
v_salary VARCHAR2(10);
BEGIN
SELECT salary
INTO v_salary
FROM employees
WHERE employee_id=101;
END;
```

В этой БД поле *salary* имеет тип NUMBER(8, 2), однако *v_salary* – это переменная типа VARCHAR2(10). *PL/SQL* автоматически преобразует числовые данные в строку символов, а затем присваивает ее символьной переменной. *PL/SQL* может автоматически выполнять преобразование между:

– символьными и числовыми типами;

– символьными типами и типами даты.

Хотя в *PL/SQL* производится неявное преобразование типов данных, при программировании рекомендуется использовать явные функции преобразования. В следующем примере это делается при помощи функции TO_CHAR:

```
DECLARE
v_salary VARCHAR2(10);
BEGIN
SELECT TO_CHAR (salary)
INTO v_salary
FROM employees
WHERE employee_id=101;
END;
```

Преимущество такого способа заключается в том, что при желании строку символов явного формата можно использовать также в функции TO_CHAR. Это облегчает понимание программы и подчеркивает преобразование типов.

Автоматическое преобразование типов может осуществляться также при обработке в *PL/SQL* различных выражений, но и для выражений применима та же рекомендация: использовать явные функции преобразования типов.

7.6. Понятие курсоров. Явные курсоры

По сути, курсор представляет собой указатель на результирующее множество в БД. Но курсор – это не просто указатель на данные в таблице. Он указывает на область памяти в глобальной области процесса (*PGA – program global area*), которая называется контекстной областью и в которой хранится следующее:

– строки, возвращенные запросом;

– число строк, обработанных запросом;

– указатель на разобранный запрос в общем пуле.

Таким образом, создается указатель на память, а не на сами данные. И так как данные выбираются в память при открытии курсора, то пользователю гарантируется согласованное представление данных на все время транзакции. Если после того, как был открыт курсор, данные были добавлены, удалены или обновлены, новые данные не отражаются в результирующем множестве курсора. Открытие курсора подобно созданию моментальной копии для данных, существующих в момент этого открытия.

Курсоры применяются для построчного доступа к данным результирующего множества. Существует два вида курсоров:

– явные – команды SELECT, явно объявленные в секции объявлений текущего блока или спецификации пакета;

– неявные – автоматически создаются в ответ на любые команды DML (SELECT, INSERT, UPDATE, DELETE, MERGE, SELECT FOR UPDATE) или SELECT … INTO в секции исполнения блока *PL/SQL*.

Также в качестве дополнительных средств работы с курсорами используют следующие приемы:

– курсорные переменные – объявленные типы, которые в одном и том же блоке *PL/SQL* могут быть ассоциированы с несколькими запросами;

– курсорные подзапросы или вложенные курсорные выражения – механизм встраивания курсоров в операторы *SQL*.

Явные курсоры. Обеспечивают самый высокий уровень управления обработкой курсоров. Явный курсор является статическим; команда *SQL* определяется в момент компиляции. Считается, что они работают с оператором SELECT, который единовременно возвращает более одной строки. Для того чтобы использовать явный курсор, его необходимо объявить, открыть, выбрать данные из него и закрыть.

Объявление явных курсоров. Для объявления явного курсора в качестве курсорной переменной в секции объявлений блока или пакета используется следующий синтаксис:

CURSOR *имя_курсора* [(*список_параметров*)]
[RETURN *возвращаемый_ тип*]
IS *запрос*
[FOR UPDATE [OF (*список_столбцов*)] [NOWAIT]];

где *имя_курсора* – любой разрешенный идентификатор; в *списке_параметров* указываются любые допустимые параметры, используемые для выполнения запроса; необязательная фраза RETURN определяет тип данных, который должен быть возвращен курсором, определенный в *возвращаемом_типе*; *запрос* – это любой допустимый синтаксис оператора SELECT; необязательная фраза FOR UPDATE блокирует записи при открытии курсора, т. е. они будут доступны другим пользователям только в режиме READ ONLY. Использование FOR UPDATE служит гарантией того, что:

– при выполнении цикла по записям курсора все они будут доступны для обновления и не будут заблокированы другим сеансом;

– данные будут соответствовать данным из контекстной области;

– если указано NOWAIT, программа будет завершаться сразу после открытия, если она не сможет установить исключительную блокировку.

Приведем примеры объявления явного курсора:

-- курсор без параметров, который возвращает названия всех отделов:

```
CURSOR cur_ departments IS SELECT department_name FROM departments;
```

-- курсор без параметров с условием, который возвращает названия отделов, где нет менеджера:

```
CURSOR cur_departments IS
SELECT department_name FROM departments
WHERE manager_id IS NULL;
```

-- курсор с параметрами или параметризованный курсор:

```
CURSOR cur_departments (v_manager_id NUMBER) IS
SELECT * FROM departments WHERE manager_id = v_manager_id;
```

-- курсор, который возвращает название отдела по его номеру:

```
CURSOR get_name(v_dep_id departments.department_id %TYPE)
RETURN departments.department_name %TYPE IS
SELECT department_name FROM departments
WHERE department_id = v_dep_id;
```

Открытие явных курсоров. Курсоры открываются в секциях выполнения или исключительных ситуаций *PL/SQL*-блока. Синтаксис оператора:

OPEN *имя_курсора* [(*значения_параметров*)];

Команда OPEN открывает явный курсор, т. е. подготавливает курсор к использованию. На стадии выполнения запрос разбирается, вычисляются значения переменных связи, строки выбираются в контекстную область и множество результатов становится доступным. Таким образом, обработка открытого курсора включает в себя такие этапы выполнения команды *SQL*, как разбор, связывание, непосредственно открытие и исполнение. В каждый момент времени в курсоре может быть только одна активная запись.

Выборка записей из курсора. Команда FETCH выбирает записи их контекстной области в переменные, чтобы ими можно было воспользоваться в программе *PL/SQL*. Эта команда действует только на текущую запись и движется по результирующему множеству, отбирая по одной записи за раз. Исключением является только использование фразы BULK COLLECT, которая выбирает все записи результирующего множества за один раз.

FETCH *имя_курсора* INTO *список_переменных или запись_PL/SQL*;

Здесь *имя_курсора* – это имя открытого курсора, а *список_переменных* – это одна или несколько, разделенных запятыми переменных, которые по количеству и типам соответствуют количеству и типам столбцов, включенных в результирующее множество.

Если в качестве приемника использовать запись_*PL/SQL*, то для ссылки на значения из записи используется следующий синтаксис:

имя_переменной.имя_столбца

Закрытие курсора. Для того чтобы закрыть явный курсор, используется следующий синтаксис:

CLOSE *имя_курсора*;

После того как все строки извлечены, курсор должен быть закрыт. Закрытие курсора позволяет механизму оптимизации памяти *PL/SQL* освободить выделенную память в контекстной области в нужный момент.

Если курсор объявляется в локальном анонимном блоке функции или процедуры, то такой курсор будет автоматически закрыт по завершении блока. Курсоры, объявленные в пакете, должны быть закрыты явно, иначе они останутся открытыми до завершения сеанса. Попытка закрытия курсора, который не был открыт, генерирует исключение *ORA*-01001: *invalid cursor*.

Для обеспечения навигации по набору активных данных курсоры чаще всего используются с обычными циклами (LOOP…END LOOP, WHILE):

```
DECLARE
v_employees employees%ROWTYPE;
CURSOR emp_cur IS SELECT * FROM employees WHERE job_id='AD_VP';
BEGIN
OPEN emp_cur;
LOOP
FETCH emp_cur INTO v_employees;
EXIT WHEN emp_cur%NOTFOUND;
DBMS_OUTPUT.PUT_LINE (v_employees.last_name);
END LOOP;
CLOSE emp_cur;
END;
```

Существует также курсорный цикл FOR, для которого не требуется явного задания операторов OPEN, FETCH, CLOSE, т. е., хотя курсор объявлен как явный, его обработкой управляет *PL/SQL*. Кроме того, в цикле FOR используется переменная, которая никогда не объявляется в секции объявления переменных и курсоров этого блока. При попытке открытия уже открытого курсора в цикле FOR *PL/SQL* генерирует исключение *ORA-06511:PL/SQL: cursor already open.*

```
DECLARE
CURSOR emp_cur IS SELECT *FROM employees WHERE job_id='AD_VP';
BEGIN
FOR v_employees IN emp_cur
LOOP
DBMS_OUTPUT.PUT_LINE (v_employees.last_name);
END LOOP;
END;
```

Атрибуты курсоров. Существуют шесть атрибутов для работы с курсорами *Oracle* (табл. 7.7).

Таблица 7.7

Атрибуты курсора

Название атрибута	Описание
%ISOPEN	Возвращает TRUE, если курсор открыт; FALSE – если курсор не открыт
%FOUND	Возвращает NULL перед первым извлечением, TRUE – при успешном извлечении записи, FALSE – если не было возвращено ни одной строки. Генерируется исключение INVALID_CURSOR в случае, если курсор не был открыт или уже закрыт
%NOTFOUND	Возвращает NULL перед первым извлечением, FALSE при успешном извлечении записи, TRUE – если не было возвращено ни одной строки. Генерируется исключение INVALID_CURSOR в случае, если курсор не был открыт или уже закрыт
%ROWCOUNT	Возвращает количество записей, извлеченных из курсора в произвольный момент времени. Генерируется исключение INVALID_CURSOR в случае, если курсор не был открыт или уже закрыт
%BULK_EXEPTIONS	Предоставляет информацию об исключительных ситуациях при использовании операций с массивами или фразы BULK COLLECT (псевдоассоциативный массив, содержащий порядковые номера обработанных записей, вызвавших исключение и соответствующие коды ошибок). Для продолжения обработки после возникновения исключения применяется параметр SAVE EXCEPTIONS
%BULK_ROWCOUNT	Также используется при операциях BULK COLLECT и предоставляет информацию о числе строк, измененных за время операции (псевдоассоциативный массив, содержащий количество записей, измененных FORALL для каждого элемента коллекции)

7.7. Неявные курсоры

Язык *PL/SQL* объявляет неявный курсор и работает с ним при каждом выполнении команды *DML* (INSERT, UPDATE, DELETE или MERGE) или команды SELECT INTO, которая возвращает одну строку из БД непосредственно в структуру данных *PL/SQL*.

Каждый раз, когда выполняется команда *SQL*, идет работа с неявными курсорами, так как каждому выполняемому оператору *DML* предоставляется контекстная область в *PGA*. В отличие от явных курсоров, неявные не надо объявлять, открывать, извлекать из них строки и закрывать, но можно использовать те же самые шесть атрибутов явных курсоров (см. табл. 7.7) в виде SQL%атрибут.

Общий синтаксис использования неявных курсоров можно представить следующими образом:

Команда_SQL
[RETURNING *значение* [, *значение*…]
INTO *переменная* [, *переменная*...]]
SQL%*атрибут*;

Инструкция RETURNING применяется в командах INSERT, UPDATE и DELETE для получения данных, измененных соответствующей командой *DML*. Применение этой инструкции позволяет избежать дополнительной команды SELECT для запроса результатов выполнения команды *DML*, например:

```
DECLARE
v_last_name employees.last_name%TYPE;
v_email employees.email%TYPE;
BEGIN
UPDATE employees SET email = 'NKOCHHAR@gmail.com'
WHERE employee_id = 101
RETURNING last_name, email INTO v_ last_name, v_email;
DBMS_OUTPUT.PUT_LINE ('У сотрудника: ' || v_ last_name || ' изменен электронный адрес на ' || v_email);
END;
```

Как отмечалось выше, для неявных курсоров можно использовать те же самые шесть атрибутов явных курсоров в виде SQL%*атрибут*, но со следующими уточнениями:

SQL%ISOPEN – всегда возвращает FALSE, поскольку курсор закрывается автоматически сразу после выполнения инструкции;

SQL%FOUND – веред выполнением инструкции возвращает NULL. Если хотя бы одна строка была вставлена или изменена, то возвращает TRUE, иначе возвращает FALSE;

SQL%NOTFOUND – перед выполнением инструкции возвращает NULL. Если хотя бы одна строка была вставлена или изменена, то возвращает FALSE, иначе возвращает TRUE;

SQL%ROWCOUNT – возвращает количество строк, обработанных курсором;

SQL% BULK_EXEPTIONS – возвращает псевдоассоциативный массив, содержащий порядковые номера обработанных записей, вызвавших исключение и соответствующие коды ошибок (относится к операциям с коллекциями);

SQL%BULK_ROWCOUNT – возвращает псевдоассоциативный массив, содержащий количество строк, которые обработаны инструкциями, выполненными в операциях пакетного связывания (относится к операциям с коллекциями).

Атрибуты явных курсоров используются в соответствии со следующим синтаксисом:

имя_курсора%атрибут

Приведем пример использования атрибута неявного курсора, для чего создадим копию таблицы:

```
CREATE TABLE employees_temp AS SELECT * FROM employees;
```

Анонимным блоком обновим данные о заработной плате сотрудников, если она менее 5000 и вернем количество записей, обработанных командой UPDATE:

```
BEGIN
  UPDATE employees_temp
    SET salary = salary * 1.05 WHERE salary < 5000;
DBMS_OUTPUT.PUT_LINE('Updated ' || SQL%ROWCOUNT || ' salaries.');
END;
```

7.8. Курсорные переменные и подзапросы

Во всех приведенных выше примерах явных курсоров рассматривались статические курсоры, связанные с одним *SQL*-оператором, который был известен при компиляции блока. Курсорная переменная может быть связана с различными операторами во время выполнения программы. Курсорные переменные аналогичны переменным *PL/SQL*, в которых могут содержаться различные значения. Однако статические курсоры аналогичны константам *PL/SQL*, так как они могут быть связаны только с одним запросом на этапе выполнения программы.

Чтобы воспользоваться курсорной переменной, ее необходимо предварительно объявить. Во время выполнения программы должна быть выделена память для хранения этой переменной, так как курсорные переменные имеют тип REF. Затем ее можно открывать, считывать и закрывать так же, как статический курсор. Обычно курсорные переменные используются внутри хранимой процедуры, которая возвращает переменную клиентской программе. Таким образом, объявление курсора может быть отделено от его реализации, например заголовок может находиться в спецификации пакета, а реализация – в теле пакета.

Объявление курсорной переменной. Курсорные переменные имеют ссылочный тип. Чтобы воспользоваться ссылочным типом, необходимо сначала объявить переменную, а затем выделить область памяти. Ссылочные типы *PL/SQL* объявляются следующим образом:

REF *тип*;

где *тип* – это предварительно определенный тип. Ключевое слово REF означает, что новый тип будет указателем на ранее определенный тип. Таким образом, тип курсорной переменной – REF CURSOR. Полный синтаксис описания типа курсорной переменной таков:

TYPE *имя_типа* IS REF CURSOR [RETURN *возвращаемый_тип*];

где *имя_типа* – это имя нового ссылочного типа, а *возвращаемый_тип* – тип записи, указывающий типы списка выбора, которые в итоге будут возвращаться курсорной переменной. Типом, возвращаемым курсорной переменной, должен быть тип записи. Запись может быть объявлена явно как определяемая пользователем или неявно при помощи %ROWTYPE. После определения ссылочного типа можно объявить переменную. Ниже приводится пример объявления различных курсорных переменных.

```
-- Описание при помощи %ROWTYPE
TYPE t_employees_ref IS REF CURSOR
RETURN employees%ROWTYPE;

-- Определяем новый тип записи
TYPE t_NameRecord IS RECORD (
first_name employees.first_name%TYPE,
last_name employees.last_name%TYPE);

-- переменную этого типа
v_NameRecord t_NameRecord;

 --и курсорную переменную, использующую этот тип записи
TYPE t_NamesRef IS REF CURSOR
RETURN t_NameRecord;

-- При помощи %TYPE можно объявить еще один тип
TYPE t_NamesRef2 IS REF CURSOR
RETURN t_NameRecord%TYPE;

-- Объявим курсорные переменные, используя созданные выше типы
v_employeesCV t_ employees_ref;
v_NameCV t_NamesRef;
```

Ограниченные и неограниченные курсорные переменные. Курсорные переменные, рассмотренные выше, являются ограниченными: они объявляются только для конкретного возвращаемого типа. Переменная должна открываться для такого запроса, список выбора которого соответствует типу, возвращаемому курсором. В противном случае возникает предопределенная исключительная ситуация ROWTYPE_MISMATCH (несоответствие типов строк). Однако в *PL/SQL* разрешается объявлять неограниченные курсорные переменные, для которых предложение RETURN отсутствует. Такая переменная может быть открыта для любого запроса. Например:

```
-- Определим неограниченный ссылочный тип
TYPE t_FlexibleRef IS REF CURSOR;
-- и переменную этого типа.
v_CursorVar t_FlexibleRef;
```

Открытие курсорной переменной для запроса. Для связи курсорной переменной с определенным оператором SELECT используется расширенный синтаксис оператора OPEN, позволяющий указать требуемый запрос:

OPEN *курсорная_переменная* FOR *оператор_select*;

где *курсорная_переменная* – это ранее объявленная курсорная переменная, а *оператор_select* – требуемый запрос. Если курсорная переменная ограничена, список выбора должен соответствовать типу, возвращаемому курсором. Иначе выдается сообщение об ошибке: ORA-6504: *PL/SQL*: *return types of result set variables or query do not match* (*PL/SQL*: не установлено соответствие между возвращаемыми типами переменных результирующего набора и запросом).

Оператор OPEN...FOR, в принципе, аналогичен оператору OPEN: анализируются переменные привязки и определяется активный набор. После выполнения OPEN...FOR можно считывать информацию из курсорной переменной. Считывание производится с помощью оператора FETCH (так же, как для обычного курсора) либо на стороне клиента.

Закрытие курсорных переменных. На сервере курсорные переменные закрываются точно так же, как статические курсоры, – при помощи оператора CLOSE. При этом освобождаются ресурсы, используемые запросом, однако память, отведенная для хранения самой курсорной переменной, освобождается не всегда, а только когда переменная выходит из области своего действия. Запрещается повторно закрывать ранее закрытые курсоры и курсорные переменные.

Существует **ряд ограничений для курсорных переменных**, таких как:

1. Нельзя объявлять курсорные переменные на уровне пакета. Однако такие переменные можно объявлять в функциях и процедурах пакетов.

2. Нельзя присваивать курсорной переменной значение NULL, а также применять операторы сравнения для проверки на равенство, неравенство или неопределенность.

3. Ни столбцы БД, ни коллекции не могут хранить курсорные переменные.

4. Нельзя применять вызовы удаленных процедур для передачи курсорных переменных с одного сервера на другой.

Курсор и результирующее множество представляют собой объекты в памяти сессии. Результирующее множество хранится до тех пор, пока на него указывает курсор. Поэтому через переменную REF CURSOR можно организовать передачу результирующих множеств данных от одного программного модуля *PL/SQL* к другому. Также переменные типа REF CURSOR позволяют скрыть незначительные изменения в запросе.

Курсорные подзапросы. Курсорные подзапросы также называют вложенными курсорными выражениями, так как они используют курсорное выражение внутри оператора SELECT языка *SQL*. Другими словами, это курсоры, используемые как выражение для столбца в списке выбора для явного курсора. Они могут использоваться со всеми определенными ранее типами курсоров, за исключением неявных. Возвращаемое значение всегда имеет тип REF CURSOR.

Курсорные выражения могут уменьшить объем ненужных данных, возвращаемый в вызывающую программу. В связи с этим их можно применять вместо приемов, требующих соединения таблиц. Курсорное выражение открывается автоматически при извлечении родительской строки, а закрывается, когда происходит одно из событий:

– вложенный курсор явно закрывается программой;

– закрывается родительский курсор;

– заново выполняется родительский курсор;

– во время извлечений родительской строки порождается исключение.

CURSOR возвращает вложенный курсор из запроса и имеет следующий синтаксис:

CURSOR(*подзапрос*)

Например, в *SQL*Plus* используем курсорное выражение в списке возвращаемых столбцов запроса:

```
SELECT department_name, CURSOR(SELECT salary, commission_pct
  FROM employees e WHERE e.department_id = d.department_id)
  FROM departments d
  ORDER BY department_name;
```

В следующем примере рассмотрим использование выражения CURSOR в качестве аргумента функции, которая может принять REF CURSOR как входной параметр.

Сначала создается функция:

```
CREATE FUNCTION f (cur SYS_REFCURSOR, mgr_hiredate DATE)
RETURN NUMBER IS
  emp_hiredate DATE;
  before number := 0;
```

```
  after number := 0;
BEGIN
  LOOP
    FETCH cur INTO emp_hiredate;
    EXIT WHEN cur%NOTFOUND;
    IF emp_hiredate > mgr_hiredate THEN after := after+1;
    ELSE before := before+1;
    END IF;
  END LOOP;
  CLOSE cur;
  IF before > after THEN RETURN 1;
  ELSE RETURN 0;
  END IF;
END;
```

Эта функция принимает в качестве параметров курсор и дату, а следующий запрос использует функцию, чтобы найти тех менеджеров, большинство сотрудников которых были наняты на работу до своего менеджера.

```
SELECT e1.employee_id, e1.last_name FROM employees e1
  WHERE f( CURSOR (SELECT e2.hire_date FROM employees e2
  WHERE e1.employee_id = e2.manager_id), e1.hire_date) = 1
  ORDER BY last_name;
```

Контрольные вопросы и задания к разделу 7

1. Какие группы типов данных выделяют в *PL/SQL*?
2. Чем отличаются символьные типы CHAR и VARCHAR2 друг от друга?
3. Какой тип хранит дату и время с точностью до 9 секунд?
4. Какие виды коллекций можно объявить в PL/SQL?
5. Как закрепить за переменной *var* тип данных поля *field* таблицы *test*?
6. Как объявить переменную, связанную с типом существующего курсора?
7. Запустите примеры программ, поясняющие работу функций и методов коллекций, дополните коллекции новыми данными.
8. Какие типы данных относятся к ссылочным?
9. Измените код анонимного блока с выводом количества строк, обновленных неявным курсором так, чтобы выводилось еще общее количество строк в таблице и количество необновленных строк.
10. Создайте и запустите на выполнение пример с использованием выражения CURSOR в качестве аргумента функции.

8. ФУНКЦИОНАЛЬНЫЕ ВОЗМОЖНОСТИ ПРОГРАММ

8.1. Условные управляющие операторы

Секция выполнения содержит код, реализующий функциональность блока. Код может содержать как любую команду *SQL*, так и любую управляющую структуру *PL/SQL*.

Секция выполнения блока *PL/SQL* состоит из одной или более логических команд. Каждая команда завершается разделителем – точкой с запятой. Физический маркер конца строки в программе *PL/SQL* игнорируется компилятором (воспринимается только как завершение однострочного комментария, который начинается символами «--»).

PL/SQL включает в себя три разновидности команды IF-THEN-ELSE, а также команду и выражение CASE. Синтаксис CASE появился только в версии *Oracle*9*i*.

IF-THEN-ELSE позволяет реализовать в программах условную логику.

1 вид: конструкция IF-THEN:

IF *условие* THEN
выполняемая команда(*-ы*);
END IF;

2 вид: конструкция IF-THEN-ELSE:

IF *условие* THEN
последовательность выполняемых команд;
ELSE
последовательность выполняемых команд;
END IF;

3 вид: конструкция IF_THEN ELSIF:

IF *условие-1* THEN
ELSIF *условие-N* THEN
команды-N;
[ELSE
else_команды;]
END IF;

Рассмотрим пример для IF-THEN-ELSIF:

```
DECLARE
x   NUMBER := 6;
BEGIN
  IF x = 3 THEN DBMS_OUTPUT.PUT_LINE ('x=3');
  ELSIF x = 5 THEN DBMS_OUTPUT.PUT_LINE ('x=5');
  ELSIF x = 10 THEN DBMS_OUTPUT.PUT_LINE ('x=10');
```

```
   ELSE DBMS_OUTPUT.PUT_LINE ('x not in (3, 5, 10)');
END IF;
END;
```

Оператор CASE. Начиная с *Oracle*9*i* поддерживаются два типа операторов CASE:

– простой оператор CASE связывает одну или несколько последовательностей операторов *PL/SQL* с соответствующим значением (последовательность для выполнения выбирается с учетом результатов вычисления выражения, возвращающего указанное значение);

– поисковый оператор (команда) CASE выбирает для выполнения одну из последовательностей операторов в зависимости от результатов вычисления списка логических условий (выполняется последовательность операторов, связанная с первым условием, результат проверки которого оказался равным TRUE).

В дополнение к операторам CASE *PL/SQL* поддерживает CASE-выражение, которое очень похоже на оператор CASE. Оно позволяет выбрать для вычисления одно или несколько выражений. В итоге получается одно значение, тогда как результатом работы оператора CASE является выполнение последовательности операторов *PL/SQL*.

И команда, и выражение CASE должны включать в себя инструкцию ELSE, которая будет выполнена в случае, если ни одна из инструкций WHEN не будет оценена как TRUE, так как в случае отсутствия совпадения *PL/SQL* выдаст ошибку.

Оператор CASE позволяет выбрать для выполнения одну последовательность команд из множества предложенных последовательностей.

```
CASE выражение-переключатель
WHEN выражение_для_значения THEN действие;
[WHEN выражение_для_значения THEN действие;...]
[ELSE действие;]
END CASE;
```

Команда CASE похожа на команду IF-THEN-ELSIF. Но в команде CASE выражение-переключатель следует непосредственно с ключевым словом CASE. Выражение вычисляется и сравнивается со значением каждой инструкции WHEN. Как только операция сравнения возвращает значение TRUE, управление передается команде, следующей за END CASE.

Если выражение-переключатель вычислено как NULL, то ему будет соответствовать только инструкция ELSE.

Инструкция WHEN NULL не будет соответствовать никакому значению, так как сервер *Oracle* выполняет проверку равенства выражений.

Максимальное количество аргументов в операторе CASE – 255, причем каждое выражение WHEN ... THEN рассматривается как два аргумента.

Перепишем предыдущий пример с IF-THEN-ELSIF в синтаксисе CASE:

```
DECLARE
x   NUMBER := 6;
BEGIN
  CASE x
    WHEN 3 THEN DBMS_OUTPUT.PUT_LINE ('x=3');
    WHEN 5 THEN DBMS_OUTPUT.PUT_LINE ('x=5');
    WHEN 10 THEN DBMS_OUTPUT.PUT_LINE ('x=10');
    ELSE DBMS_OUTPUT.PUT_LINE ('x not in (3, 5, 10)');
END CASE;
END;
```

Поисковый оператор CASE имеет следующий синтаксис:

```
CASE
  WHEN <выражение 1> THEN <операторы 1>
  WHEN <выражение 1> THEN <операторы 2>
 …
  WHEN <выражение 1> THEN <операторы N>
 [ ELSE    <операторы else> ]
END CASE;
```

Выражения CASE работают аналогично операторам CASE, но не для исполняемых операторов, а для вычисления выражений.

Простое выражение CASE позволяет выбрать для вычислений одно из нескольких выражений на основе заданного скалярного значения:

```
<простое_выражение_case> :=
 CASE <выражение>
  WHEN <результат 1> THEN <результирующее выражение 1>
  WHEN <результат 2> THEN <результирующее выражение 2>
...
  ELSE   <результирующее выражение else>
  END;
```

Поисковое выражение CASE:

```
<поисковое_выражение_case> :=
 CASE
  WHEN <выражение 1>  then <результирующее выражение 1>
  WHEN <выражение 2>  then <результирующее выражение 2>
...
  ELSE   <результирующее выражение else>
  END;
```

В отличие от команды CASE выражение CASE не содержит выражения-переключателя; вместо этого каждая инструкция WHEN содержит полное булево выражение. Выполняется первое подходящее выражение WHEN, затем управление переходит к команде, следующей за END CASE. Например:

```
DECLARE
   x   NUMBER := 6;
   y   NUMBER := 0;
BEGIN
  y:=
   CASE x
   WHEN 3  THEN 9
   WHEN 5  THEN 15
   WHEN 10 THEN 50
   ELSE 100
   END;
   DBMS_OUTPUT.PUT_LINE ('x='||x||' y='||y);
END;
```

Кроме всего перечисленного, оператор CASE можно использовать в *SQL*-запросах. Рассмотрим два примера:

```
SELECT CASE
WHEN USER = 'HR' THEN 'Приветствую владельца схемы'
WHEN USER = 'SYSTEM' THEN 'Приветствую администратора'
ELSE 'Приветствую гостя'
END NAME
FROM DUAL;
```

```
SELECT job_id, job_title,
CASE
WHEN max_salary < 5000 THEN 'младший персонал'
WHEN max_salary BETWEEN 5000 AND 10000 THEN 'АУП'
ELSE 'высшее руководство'
END "СТАТУС"
FROM jobs;
```

8.2. Команды управления последовательностью выполнения

Существует два типа команд управления последовательностью выполнения: GOTO и NULL.

GOTO выполняет безусловный переход к указанной метке.

GOTO *имя метки*;

Обычно применять команду GOTO не рекомендуется. Если она используется, то за указанной меткой должна следовать хотя бы одна команда (при необходимости этой единственной выполняемой командой может быть команда NULL).

На применение команды GOTO накладываются следующие ограничения:

– невозможен переход из команды IF, LOOP или вложенного блока;

– невозможен переход в команду IF, LOOP или вложенный блок;

– невозможен переход из одной секции команды IF в другую (из секции IF/THEN в секцию ELSE);

– невозможен переход из подпрограммы и в нее;

– невозможен переход из секции обработки исключений в выполняемую секцию блока *PL/SQL*;

– невозможен переход из выполняемой секции в секцию обработки исключений блока *PL/SQL*, хотя такой переход осуществляется командой RAISE.

В *PL/SQL* в конструкциях циклов нет такого, иногда полезного, оператора, как CONTINUE (завершить текущую итерацию в цикле и перейти к следующей) вследствие того, что выражение CONTINUE зарезервировано языком *PL/SQL* и используется для других целей. Но такую конструкцию можно эмулировать применив цикл вида LOOP EXIT WHEN END LOOP и используя весьма непопулярный, но в данном случае очень полезный оператор GOTO. Например, выведем все четные числа до 10:

```
DECLARE
s NUMBER := 0;
BEGIN
DBMS_OUTPUT.ENABLE;
LOOP
 IF (MOD(s, 2) = 1) THEN GOTO LESS;
 END IF;
DBMS_OUTPUT.PUT_LINE (TO_CHAR(s)||' is even!');
<<LESS>>
EXIT WHEN (s = 11);
s := s + 1;
END LOOP;
END;
```

NULL – это команда, которая ничего не делает.

```
NULL;
```

Команда NULL полезна, когда нужна выполняемая, но ничего не делающая команда после метки. Также она может применяться для улучшения читаемости структуры IF-THEN-ELSE. Рассмотрим пример:

```
DECLARE
```

```
cntreg INTEGER := 6;
reg INTEGER;
BEGIN
  SELECT MAX (region_id) INTO reg FROM regions;
  FOR i IN reg + 1 ..reg + cntreg LOOP
    IF i > 8 THEN
      GOTO label_exit;
    END IF;
    INSERT INTO regions (region_id, region_name) VALUES (i, 'TEST' );
  END LOOP;
<<label_exit>>
NULL;
END;
```

В этом примере, если не использовать NULL после метки <<label_exit>>, то программа *PL/SQL* сгенерирует исключение.

8.3. Команды цикла

Структура цикла позволяет многократно выполнять последовательность команд. В *PL/SQL* имеется несколько вариантов циклов:

– простой цикл (LOOP);

– условный цикл (WHILE);

– итеративный цикл (FOR), который имеет две разновидности: цикл со счетчиком и цикл по курсору.

Пути выхода из циклов:

– EXIT – безусловный выход из цикла. Используется посредством применения оператора IF;

– EXIT WHEN – выход при выполнении условия;

– GOTO – выход из цикла во внешний контекст.

Циклы могут быть дополнительно (необязательно) помечены для того, чтобы улучшить читаемость и контроль за их выполнением. Метка должна стоять непосредственно перед командой начала цикла. Например, использование меток цикла упрощает сопровождение программы, когда используется длинный цикл со вложенными в него другими циклами:

```
<<region>>
FOR reg in 1..5 LOOP
<<depatm>>
 FOR dep IN 4..20 LOOP
   IF region.reg=depatm.dep THEN DBMS_OUTPUT.PUT_LINE ('TRUE');
   END IF;
 END LOOP depatm;
END LOOP region;
```

Команда EXIT применяется для прерывания цикла и передачи управления команде, следующей за END LOOP. Синтаксис команды EXIT:

EXIT [WHEN *логическое_условие*];

Если в команду EXIT не включена инструкция WHEN, то цикл будет немедленно завершен командой. В противном случае цикл закончится, только если значение выражения «*логическое условие»* будет вычислено как TRUE. Команда EXIT является необязательной и может присутствовать в любой точке цикла.

Простой цикл. При простейшей конструкции такой цикл состоит только из команд цикла LOOP и END, между которыми должна быть заключена хотя бы одна выполняемая команда.

```
LOOP
выполняемая команда(-ы);
END LOOP;
```

Простой цикл обычно применяется, если необходимо гарантировать хотя бы однократное выполнение тела цикла. Такой цикл завершается, только когда в его теле будет выполнена команда EXIT или EXIT WHEN (или не будет обработано сгенерированное в нем исключение). Если этого не произойдет, то цикл будет выполняться бесконечно.

```
DECLARE
i   INTEGER := 0;
y   INTEGER := 10;
BEGIN
  LOOP
    IF i >= y THEN EXIT;
    END IF;
i := i + 1;
END LOOP;
DBMS_OUTPUT.PUT_LINE (i);
END;
```

Цикл WHILE. Цикл с условием, который выполняется до тех пор, пока логическое условие истинно (TRUE), и завершается, когда условие принимает значение FALSE или NULL.

```
WHILE условие
LOOP
выполняемая команда(-ы);
END LOOP;
```

Цикл WHILE применяется в тех случаях, когда заранее неизвестно, сколько раз следует повторять операции. Кроме того, цикл WHILE можно применять вместо простого цикла, если не нужно, чтобы тело цикла выполнилось даже единожды.

Цикл FOR со счетчиком. Традиционный цикл со счетчиком, в котором указывается количество повторений.

```
FOR переменная_цикла
IN [REVERSE] наименьшее_число...наибольшее_число
LOOP
выполняемая команда(-ы);
END LOOP;
```

Процессор *PL/SQL* автоматически объявляет переменную цикла с типом PLS_INTEGER. Значения *наименьшее_число* и *наибольшее_число* могут быть переменными, но вычисляются они только один раз – при первом входе в цикл. Ключевое слово REVERSE означает, что выполнение начнется со значения *наибольшее число* и будет происходить по убыванию вплоть до значения *наименьшее_число*.

При использовании цикла FOR следует придерживаться правил:

– переменную цикла можно не объявлять – она будет объявлена неявно с типом INTEGER и иметь область действия – тело цикла. Поэтому ссылаться на нее вне цикла нельзя;

– внутри цикла нельзя изменять счетчик цикла и границы диапазона;

– использовать REVERSE для обратного счета, когда счетчик уменьшается от начального значения к конечному;

– в цикле нельзя задавать шаг приращения. Счетчик всегда увеличивается на 1;

– допускается использовать операторы EXIT / EXIT WHEN.

Рассмотрим пример с диапазоном, определяемым переменной:

```
DECLARE
m   INTEGER := 3 * 2;
BEGIN
  FOR i IN 1 ..m
  LOOP
DBMS_OUTPUT.PUT_LINE ('m=' || m || ' i=' || i);
  END LOOP;
END;
```

а также пример использования обратного счета:

```
BEGIN
DBMS_OUTPUT.ENABLE (100);
 FOR i IN reverse 1..10 LOOP
```

```
DBMS_OUTPUT.PUT_LINE ('i='||i);
END LOOP;
END;
```

Цикл FOR по курсору. Цикл, определяемый явным курсором или командой SELECT. Этот тип цикла автоматически открывает курсор, выбирает все, определяемые им строки, и закрывает.

FOR *переменная_запись* IN {*имя_курсора* | (*команда*_SELECT)}
LOOP
Выполняемая команда(-ы);
END LOOP;

Процессор *PL/SQL* автоматически объявляет переменную цикла как запись типа *имя_курсора*%ROWTYPE, об этом необходимо помнить и не объявлять самим переменную с таким же именем.

Например:

```
DECLARE
-- отдел IT
vdep departments.department_id%TYPE := 60;
-- % повышения (снижения) заработной платы
pctsal   NUMBER (5, 2) := 30;
newsal employees.salary%TYPE;
BEGIN
  FOR listemp IN (SELECT emp.employee_id, emp.job_id, emp.salary,
jobs.min_salary, jobs.max_salary FROM employees emp, jobs
WHERE department_id = vdep AND emp.job_id = jobs.job_id)
  LOOP
newsal := listemp.salary * (100 + pctsal) / 100;
    IF newsal < listemp.min_salary THEN newsal := listemp.min_salary;
    ELSIF newsal > listemp.max_salary THEN newsal := listemp.max_salary;
    END IF;
DBMS_OUTPUT.PUT_LINE ( 'Сотрудник '
                    || listemp.employee_id
                    || ' старая зарплата ='
                    || listemp.salary
                    || ' новая зарплата ='
                    || ' = '
                    || newsal);
  END LOOP;
END;
```

8.4. Директивы компилятора

Ключевое слово PRAGMA позволяет давать инструкцию компилятору. В *PL/SQL* существует четыре типа директив:

– AUTONOMOUS_TRANSACTION – сообщает компилятору о том, что функция, процедура, анонимный блок *PL/SQL* верхнего уровня, объектный метод или триггер БД выполняется в собственном пространстве транзакций (рассмотрена в подразд. 8.5).

– EXCEPTION_INIT – сообщает компилятору о том, что следует сопоставить указанный номер ошибки с идентификатором, который был объявлен как EXCEPTION в текущей программе или в доступном пакете (рассмотрена в подразд. 8.6).

– SERIALLY_REUSABLE – сообщает среде выполнения, что данные пакета не должны храниться в промежутках между обращениями к нему. Применяется для снижения требования и памяти, необходимой для одного пользователя, в случае если данные пакета нужны только на протяжении одного вызова, а не всего сеанса (рассмотрена в подразд. 9.5).

– RESTRICT_REFERENCES – сообщает компилятору уровень чистоты программы или пакета. Уровень чистоты показывает, в какой степени программа свободна от возможных побочных эффектов, связанных с чтением/записью таблиц БД и/или переменных пакета (после выхода 8 версии не используется).

Директива компилятора RESTRICT_REFERENCES. Данная директива указывает уровень чистоты и имеет такой синтаксис:

PRAGMA RESTRICT_REFERENCES (*имя_программы* | DEFAULT, *уровень_чистоты*);

Ключевое слово DEFAULT применяется ко всем методам объектного типа и ко всем программам пакета.

Можно определить от одного до пяти уровней чистоты в любом порядке в списке через запятую. Уровень чистоты указывает, насколько программа или метод свободны от побочных эффектов (перечень возможных побочных эффектов и соответствующих уровней чистоты приведен в табл. 8.1).

Таблица 8.1

Уровни чистоты и побочные эффекты

Уровень чистоты	Описание	Ограничение
WNDS	Не пишет в БД	Не выполняет команды INSERT, UPDATE и DELETE
RNDS	Не читает БД	Не выполняет команду SELECT
WNPS	Не пишет в пакет	Не изменяет переменные пакета
RNPS	Не читает пакет	Не читает переменные пакета
TRUST		Не вводит объявленные ограничения, но благодаря ему компилятор считает, что они выполнены

8.5. Автономные транзакции

Появившиеся в *Oracle*8*i* автономные транзакции (*autonomous transactions*) выполняются внутри блока кода как отдельные транзакции из внешней (главной) транзакции. Изменения, осуществленные в автономной транзакции, могут быть зафиксированы или подвержены откату без фиксации/отката основной транзакции. Изменения, зафиксированные в автономной транзакции, видны главной транзакции, даже если они произошли после начала основной транзакции. Кроме того, они видны и другим транзакциям. СУБД приостанавливает основную транзакцию на время выполнения автономной.

Автономные транзакции также позволяют фиксировать и откатывать транзакции в триггерах, не вмешиваясь в работу самого триггера.

Изменения, выполненные в главной транзакции, не видны в автономной, и если главная транзакция удерживает блокировки, освобождения которых ожидает автономная транзакция, то возникает тупиковая ситуация – взаимная блокировка. Но использование параметра NOWAIT для команды UPDATE в автономной транзакции помогает минимизировать вероятность таких ситуаций. Функции и процедуры, триггеры БД, анонимные блоки *PL/SQL* верхнего уровня и объектные методы могут быть объявлены автономными при помощи директивы компилятора PRAGMA AUTONOMOUS_TRANSACTION.

Применяя автономные транзакции, необходимо следовать следующим требованиям:

– использовать директиву PRAGMA AUTONOMOUS_TRANSACTION;

– указывать COMMIT или ROLLBACK в каждой точке выхода автономной программы. В противном случае будет сгенерирована ошибка: *ORA-06519*: *active autonomous transaction detected and rolled back.*

Пример создания процедуры, использующей автономную транзакцию:

```
CREATE OR REPLACE PROCEDURE setaudit
(vtable_name IN VARCHAR2, voper_name IN CHAR,
vpk_key IN VARCHAR2, vcolumn_name IN VARCHAR2,
vold_value IN VARCHAR2, vnew_value IN VARCHAR2)
IS
PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  IF vold_value <> vnew_value OR vOper_name IN ('I','D') THEN
INSERT INTO useraudit (table_name, oper_name, pk_key, column_name, old_value, new_value, username, dateoper)
VALUES (vtable_name, voper_name, vpk_key, vcolumn_name, vold_value, vnew_value, USER, SYSDATE);
    COMMIT;
  END IF;
END;
```

При вызове процедура будет действовать независимо от вызвавшей ее родительской транзакции, т. е. она будет регистрировать не только **факт** изменений в БД, но и даже **попытку** изменения данных, даже если основная транзакция не будет зафиксирована и завершится откатом.

8.6. Секция обработки исключений

PL/SQL предоставляет пользователям гибкие и мощные средства генерации и обработки ошибок (исключений). Каждый блок *PL/SQL* может иметь собственную секцию обработки исключений, в которой исключения будут перехватываться и обрабатываться. Исключение в *PL/SQL* имеет номер ошибки и сообщение об ошибке; у некоторых исключений есть и имена.

Когда возникает (или порождается) исключение, работа операторов в выполняемой секции прекращается и управление передается непосредственно в секцию обработки исключений текущего блока *PL/SQL*. Если исключение не обработано в секции обработки исключений, то управление передается за пределы блока.

Исключительная ситуация представляет собой поименованное условие возникновения ошибки. В *PL/SQL* имеется множество заранее определенных исключительных ситуаций, соответствующих чаще всего встречающимся в *Oracle* ошибкам. Например:

– NO_DATA_FOUND (данные не обнаружены) в случае, если в результирующем множестве оператора SELECT INTO нет строк;

– TOO_MANY_ROWS (слишком много строк), когда в результирующем множестве оператора SELECT INTO более одной строки;

– DUP_VAL_ON_INDEX (повторяющееся значение в индексе) в случае, если оператор INSERT или UPDATE дублирует ключевое значение, уже находящееся в таблице;

– ZERO_DIVIDE (деление на ноль) в случае, если оператор пытается поделить число на ноль.

В *PL/SQL* включено почти 20 заранее определенных исключительных ситуаций. Когда в программе встречается заранее определенная исключительная ситуация, то управление программой автоматически передается соответствующему обработчику исключительных ситуаций, если таковой имеется.

Исключительные ситуации, определяемые пользователями, можно объявлять в разделе объявлений программы. Однако, чтобы затем установить такую исключительную ситуацию, необходимо выполнить ее явную проверку. При желании можно назначить индивидуальный номер ошибки *Oracle* исключительной ситуации, определяемой пользователем. При вызове программой указанной ошибки *Oracle* автоматически устанавливается исключительная ситуация, определяемая пользователем.

Объявление исключений. Объявить собственное исключение с типом данных EXCEPTION можно следующим образом:

DECLARE
имя_исключения EXCEPTION;

Исключение объявляется один раз в пределах блока, но вложенные блоки могут объявить исключение с тем же именем, что и в объемлющем блоке. Если объявлений несколько, то при обработке исключения объявление во внутреннем блоке имеет более высокий приоритет, чем глобальное объявление.

Все объявленные исключения имеют код ошибки 1 и выводят сообщение «User defined exception», если только не применялась директива EXCEPTION_INIT. Посредством команды PRAGMA EXCEPTION_INIT пользователь может сопоставить объявленному исключению номер ошибки:

DECLARE
имя_исключения EXCEPTION;
PRAGMA EXCEPTION_INIT (*имя_исключения*, *номер_ошибки*);

где *номер_ошибки* – это литеральная величина (переменные здесь не разрешены). Это может быть номер ошибки *Oracle* (например, –1855) или номер пользовательского диапазона от –20 000 до –20 999.

Генерация исключений. Исключение может генерироваться тремя разными способами:

– процессором *PL/SQL*,
– явной командой RAISE в коде,
– вызовом встроенной функции RAISE_APPLICATION_ERROR.

RAISE – явно генерирует исключение.

RAISE *имя_исключения*;

где *имя_исключения* – имя объявленного пользователем исключения или имя исключения, объявленного в пакете STANDARD.

Если использовать команду RAISE внутри обработчика исключений, то для повторного порождения текущего исключения имя исключения можно не указывать. За пределами секции исключений такой синтаксис некорректен.

Встроенная функция RAISE_APPLICATION_ERROR генерирует вызов исключений:

RAISE_APPLICATION_ERROR (
номер BINARY_INTEGER,
сообщение VARCHAR2,
стек_ошибок BOOLEAN DEFAULT FALSE);

где *номер* – это номер ошибки, целое число в диапазоне от –20 999 до –20 000;
сообщение – это соответствующее сообщение об ошибке;
стек_ошибок – управляет содержимым стека ошибок.

Область видимости исключений. Обработчик исключений обрабатывает или пытается обработать только исключения, порожденные в выполняемой секции блока *PL/SQL*. Исключения, генерируемые в объявлении или в секции исключений, автоматически передаются внешнему блоку. Можно ограничить область видимости исключения, поместив код *PL/SQL* в отдельный блок с собственной секцией обработки исключений.

Распространение исключений. Исключения, сгенерированные в блоке *PL/SQL*, распространяются во внешний блок, если они не обработаны в секции обработки исключений или были в ней повторно сгенерированы.

Когда возникает исключение, *PL/SQL* ищет в текущем блоке обработчик исключений, обрабатывающий соответствующую ошибку (или инструкцию WHEN OTHERS, которая описана далее). Если совпадение не обнаружено, то *PL/SQL* передает исключение во внешний блок или вызывающую программу. Распространение продолжится до тех пор, пока исключение не обработано или не передано вне самого внешнего блока обратно в вызывающую программу. В последнем случае исключение остается необрабатываемым и приводит к остановке вызывающей программы.

Если исключение порождается в блоке *PL/SQL*, оно не приводит к отказу текущей транзакции, даже если сам блок выдавал команду INSERT, UPDATE или DELETE. Для того чтобы очистить транзакцию в случае возникновения исключения, следует явно выдать команду ROLLBACK.

Однако если исключение осталось необработанным (передано за пределы самого внешнего блока), то многие среды выполняют автоматический безусловный откат всех незафиксированных изменений для сеанса.

Обработанное исключение дальше не распространяется. Для того чтобы перехватить исключение, вывести содержательное сообщение об ошибке и передать исключение во внешний блок как ошибку, его необходимо сгенерировать заново. Заново сгенерировать текущее или новое исключение позволяет описанная ранее команда RAISE.

Инструкция в обработчике исключений **WHEN OTHERS** может применяться как ловушка для всех исключений, не обрабатываемых специальными отдельными инструкциями WHEN в секции обработки исключений. Если эта инструкция присутствует, она должна быть последней в секции обработки исключений.

```
EXCEPTION
WHEN имя_исключительной_ситуации №1
THEN последовательность_операторов №1; ….

[WHEN OTHERS
                    THEN последовательность  операторов №N;]
END;
```

SQLCODE. Данная встроенная функция возвращает код ошибки *SQL* для текущего исключения. Пользователь может включить эту функцию внутрь инструкции WHEN OTHERS для обработки определенных ошибок по номеру. Директива компилятора EXCEPTION_INIT позволяет обрабатывать ошибки по имени.

SQLERRM. Данная встроенная функция возвращает сообщение об ошибке *SQL* для текущего исключения. Пользователь может включить эту функцию внутрь инструкции WHEN OTHERS для обработки определенных ошибок по номеру. Директива компилятора EXCEPTION_INIT позволяет обрабатывать ошибки по имени.

Рассмотрим пример использования функций SQLERRM и SQLCODE. Например, создадим таблицу:

```
CREATE TABLE err_test
(widget_name VARCHAR2(100),
widget_count NUMBER,
CONSTRAINT no_small_numbers CHECK (widget_count > 1000));
```

Затем заполним таблицу следующим анонимным блоком:

```
BEGIN
INSERT INTO err_test (widget_name, widget_count)
VALUES ('Athena', 2);
EXCEPTION
WHEN OTHERS THEN
IF SQLCODE = -2290 AND SQLERRM LIKE '%NO_SMALL_NUMBERS%'
THEN DBMS_OUTPUT.PUT_LINE ('widget_count is too small');
ELSE
DBMS_OUTPUT.PUT_LINE ('Exception not handled,' ||'SQLcode='
||SQLCODE);
DBMS_OUTPUT.PUT_LINE (SQLERRM);
END IF;
END;
```

Когда *PL/SQL* выполняет этот код, выводится строка

```
widget_count is too small.
```

Если ввести в этом блоке

```
INSERT INTO err_test (widget_name, widget_count) VALUES ('Athena', 'yy');
```

то в результате выполнения получим

```
Exception not handled, SQLcode=-1722
ORA-01722: invalid number
```

Контрольные вопросы и задания к разделу 8

1. Какие существуют разновидности условного управляющего оператора?
2. Выполните примеры из подразд. 8.1.
3. Какие команды управляют последовательностью выполнения хода программ?
4. Выполните пример с курсорным циклом из подразд. 8.3 для различных начальных значений процента изменения заработной платы.
5. Как и зачем объявляют автономную транзакцию в программе?
6. Выполните пример из подразд. 8.5, предварительно создав таблицу *useraudit*, для запуска процедуры использовать информацию из подразд. 9.2.
7. Какая директива сообщает среде выполнения, что данные пакета не должны храниться в течение сеанса?
8. Какая директива сопоставляет указанный номер ошибки с объявленным идентификатором?
9. Какие методы генерации исключений существуют?
10. Выполните пример с использованием функций SQLERRM и SQLCODE из подразд. 8.6.
11. В каком диапазоне чисел можно создавать пользовательские ошибки?
12. Какие ключевые слова секции исключений нужно использовать, чтобы «ловить» необработанные исключения?
13. Происходит ли откат транзакции, если исключение возникло в блоке *PL/SQL*, но было обработано в секции EXCEPTION?
14. Напишите анонимный блок, в котором происходит деление двух заданных чисел, предусмотрите в нем обработку предустановленного исключения ZERO_DIVIDE.
15. Измените код примера с курсорным циклом из подразд. 8.3 таким образом, чтобы при его запуске с несуществующим кодом отдела обрабатывалось исключение NO_DATA_FOUND.

9. ПРОГРАММНЫЕ ЕДИНИЦЫ

9.1. Общие ключевые слова процедур и функций *Oracle*

Язык программирования *PL/SQL* позволяет создавать разнообразные именованные программные единицы – контейнеры для кода. В них входят:

– процедура – программа, выполняющая одну или несколько команд;

– функция – программа, выполняющая одну или несколько команд и возвращающая значение;

– пакет – контейнер для процедур, функций и структур данных;

– триггер – программа, выполняемая в ответ на изменение в БД;

– объектный тип – *Oracle*-версия класса в объектно-ориентированном языке; элементами объектных типов могут быть процедуры и функции. Объектный тип может содержать в своих методах код *PL/SQL*.

Следующие ключевые слова в качестве общих атрибутов могут применяться при определении процедуры или функции.

– OR REPLACE – предназначено для пересоздания существующей программной единицы с сохранением ее привилегий.

– AUTHID – определяет, с какими привилегиями будет выполняться программа, и задает имена, например от имени владельца объекта (DEFINER) или пользователя, выполняющего функцию (CURRENT_USER). До версии *Oracle*8*i* только стандартные пакеты DBMS_SQL и DBMS_UTILITY выполнялись от имени текущего пользователя. Значение AUTHID по умолчанию – DEFINER. Появилось в *Oracle*8*i*.

Хранимый *SQL* поддерживает две модели назначения привилегий в процессе выполнения. По умолчанию *Oracle* устанавливает привилегии владельца программы, т. е. пользователя, создавшего ее. При таком способе привилегии должны быть предоставлены пользователю напрямую – он не может наследовать их от роли.

Во втором варианте пользователь, вызывающий программу, делает это со своими собственными привилегиями. Анонимные блоки *PL/SQL* всегда выполняются с правами текущего пользователя. Чтобы создаваемая программа выполнялась с привилегиями вызвавшего ее пользователя, нужно объявить ее с ключевыми словами AUTHID CURRENT_USER.

– AGGREGATE USING – требуется для агрегатных функций. Указывает серверу *Oracle*, что функция обрабатывает группу строк и возвращает единственный результат. Встроенная функция AVG представляет собой агрегатную функцию. Появилось в *Oracle*9*i*.

– DETERMINISTIC – требуется для индексов по ключу-функции (*function-based index*). Функция является детерминированной (DETERMINISTIC), если она всегда возвращает одно и то же значение при вызове с одинаковыми параметрами. Детерминированные функции не ссылаются на переменные пакета или

БД. Встроенная функция INITCAP является детерминированной, а SYSDATE – нет. Появилось в *Oracle*8*i*.

– PARALLEL_ENABLED [(PARTITION *входной_параметр* BY {ANY |HASH |RANGE})] – сообщает, что функция безопасна для параллельного выполнения. Инструкция PARTITION BY доступна только для функций, имеющих входной параметр REF CURSOR. Применяется для табличных функций и указывает оптимизатору, каким образом можно секционировать входные данные. Появилось в *Oracle*8*i*.

– PIPELINED – применяется для табличных функций. Сообщает серверу *Oracle*, что функция может начать возвращать данные по мере их генерирования вместо того, чтобы вернуть все данные сразу после завершения обработки. Появилось в *Oracle*9*i*.

9.2. Создание и выполнение процедур

Процедура – это программная единица или модуль, который выполняет одну или более команд и может получать или возвращать значения посредством списка параметров.

Синтаксис создания процедуры следующий:

```
CREATE [OR REPLACE] PROCEDURE имя
                [(параметр [, параметр])]
                [AUTHID {CURRENT_USER | DEFINER}]
[DETERMINISTIC]
{IS | AS}
секция_объявления;
BEGIN
секция_выполнения;
[EXCEPTION
секция_исключений;]
END [имя];
```

Общие ключевые слова: AGGREGATE USING, AUTHID, DETERMINISTIC, PARALLEL_ENABLE и PIPELINED.

Если подпрограмма создается при помощи команды CREATE OR REPLACE, она сохраняется в словаре данных, причем хранится не только исходный текст подпрограммы, но и скомпилированная форма, которая называется *p*-кодом (*p-code*). В *p*-коде содержатся все обработанные ссылки подпрограммы, а исходный текст преобразуется в вид, удобный для чтения системой поддержки *PL/SQL*. При вызове подпрограммы *p*-код считывается с диска и выполняется. Считанный *p*-код сохраняется в разделяемой области пула системной глобальной области (*SGA*), где при необходимости к нему могут обращаться несколько пользователей. Как и все другие компоненты разделяемого пула, *p*-код удаляется из него с течением времени в соответствии с алгоритмом *LRU* (*Least Recent Used* –

использовавшийся раньше всех). *Oracle* для кеширования используется модифицированный алгоритм *LRU*. *LRU*-кеш – это буфер в памяти для быстрого доступа к часто используемым элементам, он представляет из себя двусвязный список, в начале которого наиболее часто используемые элементы, а в конце – редко.

P-код аналогичен объектному коду, генерируемому компиляторами других языков третьего поколения, или байт-коду *Java*, который считывается исполняющей системой *Java*. В *p*-коде содержатся обработанные ссылки на объекты (это свойство раннего связывания), поэтому выполнение *p*-кода является сравнительно недорогой (нересурсоемкой) операцией.

Напомним, что при указании типа параметра в заголовке процедуры дается ссылка на сам тип, без указания размерности.

В разделе локальных объявлений описываются локальные переменные, константы, курсоры, область действия которых определяется рамками процедуры.

Вызов процедуры осуществляется по ее имени. Значения для параметров процедуры могут передаваться либо по позиции параметра в списке параметров, либо по имени параметра. Список значений размещается в круглых скобках вслед за именем процедуры. Когда процедура не имеет параметров, то сразу после имени ставится «;».

В качестве примера рассмотрим процедуру по изменению заработной платы указанному сотруднику:

```
CREATE OR REPLACE PROCEDURE change_salary
(empId IN NUMBER, newsal IN NUMBER) IS
maxsal jobs.max_salary%TYPE;
minsal jobs.min_salary%TYPE;
sal employees.salary%TYPE;
job_ VARCHAR2(10);

BEGIN
  SELECT job_id, salary
  INTO job_, sal
  FROM employees WHERE employee_ID = empId;

  IF sal<>newsal THEN
    SELECT max_salary, min_salary INTO maxsal, minsal
      FROM JOBS where JOB_ID = (SELECT job_id FROM employees
WHERE employee_ID =empId);
    UPDATE employees set salary = newsal WHERE EMPLOYEE_ID = empId;
    COMMIT;
    DBMS_OUTPUT.PUT_LINE ('Сотрудник '||empID||': старая зарплата = '||
sal||', новая зарплата = '||newsal);
  END IF;
END;
```

Выполним процедуру. Как отмечалось выше, для вызова процедуры из *SQL*Plus* можно воспользоваться командой EXECUTE или CALL, а также всегда и везде ее можно выполнить используя анонимный блок:

```
EXECUTE change_salary (105, 8000);
```

Можно выполнить процедуру анонимным блоком с передачей параметров по имени:

```
BEGIN
change_salary (newsal => 9350, empId => 105);
END;
```

Процедуры нельзя применять для выполнения запросов, потому что они не возвращают результат.

9.3. Создание и вызов функций

Функция – это программная единица или модуль, который выполняет одну или более команд и возвращает значение при помощи инструкции RETURN. Как и процедуры, функции могут получать одно или более значений через список параметров. Функции должны содержать в секции выполнения хотя бы одну команду RETURN. Инструкция RETURN в заголовке функции задает только тип данных возвращаемого значения.

Синтаксис определения функции имеет вид

```
CREATE [OR REPLACE] FUNCTION имя
[(параметр [, параметр])]
RETURN возвращаемый_тип_данных
[AUTHID {CURRENT_USER | DEFINER}]
[DETERMINISTIC]
[PARALLEL_ENABLED]
[PIPELINED]
[AGGREGATE USING]
{IS | AS}
[секция_объявления;]
BEGIN
секция_выполнения;
[EXCEPTION
секция_исключений;]
END [имя];
```

Возвращаемое функцией значение принадлежит к определенному типу данных. В отличие от процедур вызов функции всегда является частью оператора, т. е. включается в выражение либо служит в качестве значения по умолчанию. Функции, определенные пользователем, могут использоваться в *SQL*-выражениях.

Функции играют важную роль в обеспечении модульного подхода. К примеру, реализация отдельного бизнес-правила или формулы должна помещаться в функцию. Любой запрос, возвращающий одну строку (неявный курсор), может также объявляться внутри функции для его повторного использования. Для примера рассмотрим функцию, которая возвращает количество отработанных сотрудником лет и месяцев:

```
CREATE OR REPLACE FUNCTION get_time_work (
empid employees.employee_id%TYPE,
dateval DATE DEFAULT SYSDATE)
RETURN VARCHAR2
IS
RESULT INTERVAL YEAR TO MONTH;
d employees.hire_date% TYPE;
BEGIN
SELECT hire_date INTO d FROM employees WHERE employee_id = empid;
RESULT := (dateval - d) YEAR TO MONTH;
RETURN RESULT;
END;
```

Параметр *dateval* описан ключевым словом DEFAULT, что позволяет при обращении к функции не задавать его значение, если значение по умолчанию устраивает.

Вызов функции может осуществляться в любом месте исполняемого оператора, где возможно использование выражения того же типа, что и возвращаемый функцией результат, следовательно, вызвать функцию можно следующими способами:

– в команде присваивания:

```
DECLARE
time INTERVAL YEAR TO MONTH;
BEGIN
time := get_time_work (108);
DBMS_OUTPUT.PUT_LINE (time);
END;
```

– при задании значения по умолчанию:

```
DECLARE
time INTERVAL YEAR TO MONTH DEFAULT get_time_work (108);
BEGIN
DBMS_OUTPUT.PUT_LINE (time);
END;
```

– в логическом выражении:

```
DECLARE
time INTERVAL YEAR TO MONTH;
BEGIN
IF get_time_work (107) < INTERVAL '10-00' YEAR TO MONTH
THEN time:=get_time_work (107);
DBMS_OUTPUT.PUT_LINE (time);
END IF;
END;
```

– в команде *SQL*:

```
SELECT employee_id, first_name, get_time_work (employee_id)
FROM employees
WHERE (sysdate - hire_date) YEAR TO MONTH > get_time_work (114);
```

– как аргумент в списке параметров другой программной единицы. Например, так:

```
BEGIN
change_hire_date (105, get_time_work (107));
END;
```

где *change_hire_date* – гипотетическая процедура изменения даты приема на работу сотрудника.

Вызов функций *PL/SQL* в *SQL*. Как было отмечено, хранимые функции могут вызываться из команд *SQL* подобно тому, как вызываются встроенные числовые или символьные функции, такие как, например, DECODE, NVL или RTRIM. При осуществлении таких вызовов необходимо помнить о ряде мер предосторожности и ограничений.

Предложенный ниже формат применяется для вызова собственной функции из *SQL*. Такая возможность позволяет «переделать» *SQL* в соответствии с требованиями конкретного приложения.

Синтаксис:

[*имя_схемы*.] [*имя_пакета*.] *имя_функции* [*@db_link*]
[*список параметров*]

где *имя_схемы* – это пользователь/владелец функции или пакета (может отсутствовать);

имя_пакета – это пакет, содержащий вызываемую функцию (может отсутствовать);

имя_функции – имя вызываемой функции;

@db_link – имя канала связи с удаленной БД, содержащей функцию (необязательно);

список_параметров – это список параметров функции (необязателен).

К вызову хранимых функций в *SQL* предъявляется ряд требований:

1. Все параметры должны быть входными, параметры IN OUT и OUT не разрешены.

2. Типы данных параметров функции и возвращаемый тип (RETURN) должны быть совместимы с типами данных БД. Аргументы и возвращаемые значения не могут относиться к типу BOOLEAN, типу пользовательских записей, быть ассоциативным массивом и т. д.

3. Для передаваемых функции параметров должно использоваться позиционное представление; представление по имени не поддерживается.

4. Функция должна храниться в БД, а не в локальной программе.

9.4. Дополнительные возможности процедур и функций

Табличные функции – это функции, принимающие на входе коллекцию или REF CURSOR (набор строк) и возвращающие на выходе коллекцию записей (набор строк). Табличные функции появились в *Oracle*8*i*. Начиная с *Oracle*9*i* существует возможность применять команду PIPE ROW для идентификации входного и выходного потоков. Такая организация потоков позволяет организовывать конвейер из нескольких функций, избегая необходимости хранения промежуточных таблиц. Табличные функции обычно появляются в инструкции FROM запроса.

Создадим функцию, которая возвращала бы список сотрудников отдела, *id* которого передается в качестве параметра.

Для начала нам нужно описать тип данных, возвращаемый таблицей.

```
-- тип данных строки, возвращаемой GetEmployees
TYPE rowGetEmployees IS RECORD (
l_empinfo employees.first_name%TYPE,
l_depinfo departments.department_name%TYPE);
```

Далее необходимо создать таблицу из строк типа rowGetEmployees:

```
TYPE tblGetEmployees IS TABLE OF rowGetEmployees;
```

Переменную этого типа будет возвращать функция:

```
FUNCTION GetEmployees
(prm_depid NUMBER DEFAULT NULL)
RETURN tblGetEmployees
PIPELINED;
```

Если параметр не передан, будет возвращаться список всех сотрудников. Атрибут PIPELINED означает, что функция является конвейерной, результат возвращается клиенту немедленно при вызове директивы PIPE ROW, поэтому оператор RETURN необязателен. Фактически по результирующему набору из запроса в теле функции проходит курсор, который при каждой итерации добавляет в запись текущую строку.

Поместим типы данных и функцию в пакет:

```
CREATE OR REPLACE PACKAGE p_my1 IS
TYPE rowGetEmployees IS RECORD (
l_empinfo employees.first_name%TYPE,
l_depinfo departments.department_name%TYPE);
TYPE tblGetEmployees IS TABLE OF rowGetEmployees;
FUNCTION GetEmployees
(prm_depid NUMBER DEFAULT NULL)
RETURN tblGetEmployees
PIPELINED;
END p_my1;
```

```
CREATE OR REPLACE PACKAGE BODY p_my1 IS
FUNCTION GetEmployees
(prm_depid NUMBER DEFAULT NULL)
RETURN tblGetEmployees
PIPELINED IS
BEGIN
IF prm_depid IS NULL THEN
FOR curr IN
(SELECT emp.first_name, dep.department_name FROM departments dep
INNER JOIN employees emp ON dep.department_id = emp.department_id
) LOOP
PIPE ROW (curr);
END LOOP;
ELSE
FOR curr IN( SELECT emp.first_name, dep.department_name
FROM departments dep INNER JOIN
employees emp ON dep.department_id = emp.department_id
WHERE dep.department_id = prm_depid
) LOOP
PIPE ROW (curr);
END LOOP;
END IF;
END GetEmployees;
END p_my1;
```

Осуществляем вызов:

```
SELECT * FROM TABLE(p_my1.GetEmployees);
```

```
SELECT * FROM TABLE(p_my1.GetEmployees(10));
```

Локальные программы – это процедуры или функции, определенные в секции объявления блока *PL/SQL*. Объявление локальной программы должно

находиться в конце секции объявлений после всех объявлений типов, записей, курсоров, переменных и исключений.

На программу, определенную в секции объявлений, можно ссылаться только внутри секций выполнения и исключений того же блока. Она не определена вне данного блока.

Использование локальных модулей уменьшает объем кода и улучшает читабельность программы. Перепишем процедуру по изменению заработной платы сотрудников с использованием локальных модулей:

```
CREATE OR REPLACE PROCEDURE change_salary2
(empid IN number, newsal IN number) IS
TYPE emp_inf_type IS RECORD (
  job_id employees.JOB_ID%Type :=0,
  salary employees.SALARY%Type :=0,
  min_sal employees.SALARY%Type :=0,
  max_sal employees.SALARY%Type :=0);

empinf emp_inf_type;

FUNCTIONgetinf (emp_id in NUMBER)
RETURN  emp_inf_type
IS
result emp_inf_type;
BEGIN
  SELECT emp.job_id, emp.salary, jobs.min_salary, jobs. max_salary
INTO result FROM employees emp, jobs
WHERE employee_id = empid  AND emp.job_id = jobs.job_id;
RETURN result;
END getinf;

 FUNCTION testsalary (empinf IN EMP_INF_TYPE, newsal IN Number)
 RETURN BOOLEAN IS
 BEGIN
IF newsal < empinf.min_sal OR newsal > empinf.max_sal THEN
RETURN FALSE;
  ELSE
RETURN TRUE;
END IF;
END testsalary;
BEGIN
empinf := getinf (empid);
IF testsalary(empinf, newsal) THEN
UPDATE employees SET salary = newsal WHERE employee_id = empid;
COMMIT;
DBMS_OUTPUT.PUT_LINE ('Сотрудник '||empid||': старая зарплата = '||
```

```
empinf.salary ||', новая зарплата = '||newsal);
ELSE
DBMS_OUTPUT.PUT_LINE ('Сумма вне рамок должностного оклада');
   END IF;
END;
```

В примере используются две локальные функции: *getinf*, которая возвращает в переменную *empinf* информацию о сотруднике и *testsalary*, которая проверяет, входит ли новая зарплата в допустимые по должности рамки. Локальные программы могут быть перегружены, при этом действуют те же правила, что и для перегруженных программ пакетов.

Перегрузка программ. *PL/SQL* позволяет определить внутри любой секции объявлений (в том числе в спецификации или теле пакета) несколько программ с одинаковыми именами. Такой прием называется *перегрузкой*. Если две или более программ имеют одно и то же имя, они должны отличаться чем-то другим, чтобы компилятор мог определить, которую из них следует использовать.

Рассмотрим пример перегруженных программ в одной из спецификаций встроенного пакета *Oracle*:

```
PACKAGE DBMS_OUTPUT
IS
PROCEDURE PUT_LINE (a VARCHAR2);
PROCEDURE PUT_LINE (a NUMBER);
PROCEDURE PUT_LINE (a DATE);
END;
```

Все процедуры PUT_LINE идентичны во всем, кроме типа данных параметра. Для компилятора такого различия достаточно.

Для успешной перегрузки необходимо, чтобы было выполнено хотя бы одно из приведенных ниже условий:

– наборы параметров должны отличаться по типам данных (числовой, строковый, дата/время, логический);

– должны отличаться типы программ (можно создать функцию и процедуру с одинаковыми именами и идентичными списками параметров);

– количество параметров должно быть разным.

Программы не удастся перегрузить, если:

– отличаются только типы данных в инструкции RETURN для функций;

– типы данных параметров относятся к одному семейству (CHAR и VARCHAR2, NUMBER и INTEGER и т. д.);

– различаются только режимом использования параметров.

9.5. Пакеты

Пакет (*package*) – это группа элементов кода *PL/SQL*. Пакет может включать в себя такие элементы, как процедуры, функции, константы, переменные, курсоры, имена исключений и команды объявления типов TYPE (для ассоциативных массивов, записей, REF CURSOR и т. д.).

Пакет может состоять из двух частей: спецификации и тела.

Спецификация пакета имеет следующий синтаксис:

CREATE [OR REPLACE] PACKAGE *имя_пакета*
[AUTHID {CURRENT_USER | DEFINER}]
{IS | AS}
[*определения публичных типов,*
объявления публичных переменных, типов и объектов,
объявления исключений,
прагмы,
объявления курсоров, процедур и функций,
заголовки процедур и функций]
END [*имя_пакета*];

В спецификации перечисляются все объекты, которые доступны для общего использования в приложениях. Кроме того, спецификация пакета предоставляет всю информацию, необходимую разработчику для использования объектов пакета; можно сказать, что спецификация пакета представляет собой его *API*. Спецификация необходима всегда. В случае если спецификация или пакет не содержит никаких процедур и функций и нет необходимости в закрытом коде, то тело пакета не требуется.

Тело пакета имеет следующий синтаксис:

CREATE [OR REPLACE] PACKAGE BODY *имя_пакета*
{IS | AS}
[*определения закрытых типов,*
объявления закрытых переменных, типов и объектов,
полные определения курсоров,
полные определения процедур и функций]
[BEGIN
выполняемые_команды;
[EXCEPTION
обработка исключений;]]
END [*имя_пакета*];

Тело пакета содержит весь код, необходимый для реализации процедур, функций и курсоров, перечисленных в спецификации, а также все закрытые объекты (если такие имеются), доступные только другим элементам, определенным в данном пакете, и необязательную секцию инициализации.

Объявления из спецификации не могут быть повторены в теле пакета. Секции выполнения и обработки исключений необязательны для тела пакета. Само тело пакета может содержать выполняемую секцию, которая следует за объявлением процедур и функций внутри тела пакета. Если секция выполнения присутствует, то она вызывает *секцию инициализации (initialization section)*, выполняемую всего один раз – при первом обращении к какому-либо элементу пакета в рамках сеанса.

Спецификация пакета должна быть скомпилирована раньше его тела. При назначении привилегии EXECUTE для пакета другой схеме или всем пользователям (PUBLIC) доступ предоставляется только к спецификации, тело остается скрытым.

Использование пакетов дает следующие преимущества:

– Модульность. Логически взаимосвязанные программные структуры собираются в один модуль.

– Сокрытие информации. Возможность решить, какие конструкции будут общедоступными (видимыми и доступными для всех), а какие – частными (скрытыми и недоступными). Пакет скрывает определения частных конструкций, поэтому в случае изменения определения это отразится только на пакете, а не на приложении. Кроме того, скрывая от пользователей подробности реализации, защищается целостность пакета.

– Дополнительные функциональные возможности. Пакетные общедоступные переменные и курсоры остаются неизменными до конца сеанса. Следовательно, они могут совместно использоваться всеми подпрограммами, выполняемыми в данном окружении. Кроме того, они позволяют хранить данные на протяжении нескольких транзакций, не помещая их в БД.

– Повышение производительности. Когда какая-нибудь подпрограмма из пакета вызывается в первый раз, в память загружается весь пакет. Следовательно, последующие вызовы взаимосвязанных подпрограмм не требуют дополнительных операций ввода/вывода. Пакетные подпрограммы предотвращают также каскадное воздействие на зависимые единицы и позволяют таким образом избежать ненужной компиляции.

– Перегрузка. Пакеты позволяют перегружать (*overload*) процедуры и функции. Это означает, что в одном и том же пакете можно создать несколько одноименных подпрограмм, каждая из которых принимает параметры различного числового или иного типа данных.

Рассмотрим пример пакета:

```
CREATE OR REPLACE PACKAGE  time_pkg  IS
FUNCTION GetTimeStamp RETURN DATE;
PRAGMA RESTRICT_REFERENCES (GetTimeStamp, WNDS);
PROCEDURE ResetTimeStamp;
END time_pkg;
```

```
CREATE OR REPLACE PACKAGE BODY time_pkg IS
StartTimeStamp DATE :=SYSDATE;
-- StartTimeStamp – это данные пакета

FUNCTION GetTimestamp RETURN DATE IS
BEGIN
RETURN StartTimeStamp;
END GetTimeStamp;

PROCEDURE ResetTimeStamp IS
BEGIN
StartTimeStamp:=SYSDATE;
END ResetTimeStamp;
END time_pkg ;
```

Любое изменение заголовка делает тело пакет недействительным (*invalid*). Поэтому следует помнить, что необходимо перекомпилировать тело пакета после любого изменения в его заголовке. Если не требуется вносить в тело какие-либо изменения, то это можно сделать так:

```
ALTER PACKAGE test_pkg COMPILE;
```

Обращение к элементам пакета. На элементы, объявленные в спецификации, из вызывающего приложения ссылаются при помощи точечной нотации:

имя_пакета.элемент_пакета

Например, пакет DBMS_OUTPUT содержит процедуру с именем PUT_LINE, и обращение к ней будет выглядеть следующим образом:

```
BEGIN
DBMS_OUTPUT.PUT_LINE ('Эта строка – параметр');
END;
```

Данные пакета. Структуры данных, объявленных в спецификации или теле пакета, но вне каких бы то ни было процедур и функций, называются данными или переменными пакета. Областью действия переменных пакета является весь сеанс, для них не существует границ, поэтому они по сути своей являются глобальными. Они инициализируются только один раз – при первом обращении к пакету

Когда вызывается какая-либо составляющая пакета, *Oracle* загружает пакет в память, где он остается все то время, пока пользователь соединен с БД. При наличии нескольких сеансов переменные пакета и их значения становятся разделяемыми; следует также заметить, что обращение к объектам, помещенным в пакет, в последующих сеансах может происходить быстрее.

Выполняемая секция пакета обычно используется для инициализации локальных переменных пакета. Выполняется эта секция один раз при загрузке пакета, а следовательно, ее не стоит использовать для каких-либо повторяющихся действий, поскольку обратиться к ней более одного раза все равно не удастся.

При работе с данными пакета не забывайте о следующем:

– на состояние переменных пакета не влияют COMMIT и ROLLBACK;

– объявленный в пакете курсор – глобальный. Он остается открытым до тех пор, пока не будет закрыт явно или до конца сеанса;

– хорошей практикой следует считать скрытие структур данных в теле пакета и создание программ для чтения и записи этих данных.

Директива компилятора SERIALLY_REUSABLE. Если необходимо, чтобы данные пакета существовали только на протяжении выполнения функции или процедуры пакета, но не между вызовами в текущем сеансе, то вы можете сэкономить память, используя директиву компилятора SERIALLY_REUSABLE. Ее необходимо включить как в спецификацию, так и в тело пакета. После каждого вызова *PL/SQL* будет закрывать курсоры и освобождать занятую пакетом память.

Этот прием применим только для больших сообществ пользователей, выполняющих одну и ту же программу. Обычно требования к оперативной памяти сервера БД вырастают линейно в зависимости от количества пользователей. Если указать директиву SERIALLY_REUSABLE, этот рост может замедлиться за счет того, что рабочие области пакета хранятся в пуле *SGA* и совместно расходуются всеми пользователями.

9.6. Триггеры

Триггер (*trigger*) – это программа, выполняемая в ответ на изменение данных или на определенное событие, происходящее в БД. Существует предопределенный набор сообщений, которые могут быть связаны с триггерами, что позволяет объединять пользовательскую обработку с обработкой, которую выполняет сама СУБД. Запускает или выполняет триггер инициирующее событие.

Триггеры включаются при создании (посредством команды CREATE TRIGGER) и могут быть отключены двумя способами при помощи команд ALTER TRIGGER или ALTER TABLE:

ALTER TRIGGER *имя_триггера* {ENABLE | DISABLE};
ALTER TABLE *имя_таблицы* {ENABLE | DISABLE} ALL TRIGGERS;

Нельзя создавать триггеры для объектов, принадлежащих пользователю SYS.

Можно выделить следующие группы триггеров:

– *DML*-триггеры применяются для проверки внесенных в таблицы изменений, которые запускаются на события: вставки, обновления, удаления данных из таблиц.

– *DDL*-триггеры запускаются при выполнении *DDL*-инструкций, т. е. при создании, модификации, удалении таблиц или других объектов БД.

– Триггеры уровня БД, которые отслеживают события, касающиеся работы сервера БД, такие как его остановка, запуск, возникновение ошибок *Oracle*, а также для отслеживания пользовательских событий, таких как LOGON и LOGOFF.

– Триггеры INSTEAD OF – замещающие триггеры, которые запускаются непосредственно перед выполнением вставки, обновления, удаления и определяют, какие операции необходимо выполнить вместо соответствующей операции. Управляют операциями над представлениями (*view*), но не над таблицами и позволяют преобразовывать необновляемые представления в обновляемые представления, задавая их поведение.

Общий синтаксис команды CREATE TRIGGER:

CREATE [OR REPLACE] TRIGGER *имя_триггера*
{BEFORE | AFTER | INSTEAD OF} *инициирующее_событие*
ON
[NESTED TABLE *столбец_вложенной_таблицы* OF *представление*]
| *ссылка_на _таблицу_или_представление* | DATABASE [*инструкция_ссылок*]
[REFERENCING NEW AS NEW1 OLD AS OLD1]
[FOR EACH ROW [WHEN *условие_триггера*]]
[DECLARE ...]
BEGIN
исполняемые_операторы;
[EXCEPTION
обработка_исключений;]
END [*имя_триггера*];

Ключевые слова:

BEFORE | AFTER – триггеры могут запускаться до (BEFORE) или после (AFTER) инициирующего события. Триггер AFTER считается более эффективным, чем триггер BEFORE.

INSTEAD OF – обычно используется с необновляемыми по своей сути представлениями для того, чтобы разрешить обновление базовых таблиц представления при помощи команд INSERT, UPDATE или DELETE.

инициирующее_событие – одно из перечисленных ниже событий:

– INSERT – срабатывает при каждом добавлении строки в таблицу или представление, заданные параметром *ссылка_на _таблицу_или_представление*;

– UPDATE – срабатывает при каждом изменении таблицы или представления, заданного параметром *ссылка_на _таблицу_или_представление* посредством UPDATE. UPDATE-триггеры могут дополнительно содержать инструкцию OF для того, чтобы запретить срабатывание при обновлении определенных (указанных в этой инструкции) столбцов;

– DELETE – срабатывает при каждом удалении строки из таблицы или представления, переданного параметром *ссылка_на_таблицу_или_представление*. Не срабатывает в случае удаления командой TRUNCATE;

– CREATE – срабатывает при каждом добавлении нового объекта БД посредством команды CREATE;

– ALTER – срабатывает при каждом добавлении нового объекта БД посредством команды ALTER;

– DROP – срабатывает при каждом добавлении нового объекта БД посредством команды DROP;

– SERVERERROR – срабатывает при записи серверного сообщения об ошибке. Для таких событий разрешены только триггеры AFTER;

– LOGON – срабатывает при создании сеанса (подключении пользователя к БД). Для таких событий разрешены только триггеры AFTER;

– LOGOFF – срабатывает при закрытии сеанса (отключении пользователя от БД). Для таких событий разрешены только триггеры BEFORE;

– STARTUP – срабатывает при открытии БД. Для таких событий разрешены только триггеры AFTER;

– SHUTDOWN – срабатывает при закрытии БД. Для таких событий разрешены только триггеры BEFORE.

В контексте инициирующих событий CREATE, ALTER, DROP объекты понимаются как таблицы или пакеты (все то, что находится в представлении словаря данных ALL_OBJECTS). Такой триггер может применяться к одной схеме или ко всей БД.

инструкция_ссылок разрешена только для таких событий, как INSERT, UPDATE, DELETE. Позволяет дать имя не по умолчанию для старой и новой псевдозаписи. Эти псевдозаписи обеспечивают для программы видимость значений, которые были до обновления БД и которые будут после обновлений БД. Применяются для триггеров, действующих на уровне строк. Такие записи определяются как записи %ROWTYPE с той лишь разницей, что запрещены ссылки на столбцы типа LONG и LONG RAW. Для них применяется точечная нотация, в теле триггера они предваряются двоеточием. В отличие от других записей для полей разрешено только индивидуальное присваивание. Для триггеров INSERT все старые поля содержат NULL, для триггеров DELETE все новые поля содержат NULL.

REFERENCING позволяет изменять название псевдозаписи при создании триггера. Для манипулирования данными в триггере доступны так называемые псевдозаписи NEW и OLD – структуры данных, аналогичные типу RECORD – запись *PL/SQL*. Запись NEW содержит значения полей после внесения изменений, запись OLD – значения полей до внесения изменений. Соответственно, структура NEW доступна в триггерах, обрабатывающих операции вставки и изменения, а структура OLD – изменения и удаления. При обращении к полям псевдозаписи в теле триггера всегда записывается двоеточие, например :OLD.JOB_ID. Псевдозаписи :NEW, :OLD в триггере никогда не объявляются.

FOR EACH ROW – означает, что триггер действует на уровне строк. Такие триггеры запускаются по одному разу для каждой обработанной строки. По умолчанию триггеры действуют на уровне команды, т. е. запускаются по одному для каждой инициирующей команды.

Триггеры уровня строки имеют программный доступ к значениям записи до и после изменения данных через псевдозаписи. По умолчанию имена этих псевдозаписей :NEW и :OLD. Эти записи подобны записям, определенным с помощью атрибута %ROWTYPE, но отличаются тем, что в них нельзя ссылаться на столбцы типа LONG и LONG ROW. Для псевдозаписей (в отличие от обычных записей) нельзя использовать агрегатное присваивание – значение полям можно назначать только индивидуально.

Обозначения :NEW и :OLD, предшествующие имени поля, также называют квалификаторами.

В триггерах INSERT значения всех полей старой записи – NULL, а в триггерах DELETE значения всех полей новой записи – NULL.

WHEN *условие_триггера* – задает условие, которое должно быть выполнено, для того чтобы триггер сработал. В условии для запуска триггера не могут применяться хранимые процедуры и функции.

Триггеры *DML*. Триггеры *DML* отслеживают события и операции, производимые над данными. Это наиболее распространенный вид триггеров. *DML*-триггер может запускаться:

– до выполнения операции (триггер BEFORE) или после выполнения операции (триггер AFTER). Триггеры BEFORE используются в случае, когда надо проверить возможность выполнения операции, определить значения полей;

– один раз для всей инструкции *SQL* (триггер уровня инструкции) или отдельно для каждой изменяемой записи (триггер уровня записи или строчный триггер (FOR EACH ROW)). В этом случае, например, при обновлении 100 строк триггер будет запускаться и срабатывать 100 раз.

Триггеры принимают участие в транзакциях. Если в триггере инициируется исключение, то будет выполнен откат соответствующей транзакции. Если триггер сам изменяет данные, то все изменения становятся частью общей транзакции. В триггере нельзя применять операторы управления транзакциями COMMIT, ROOLBACK и SAVEPOINT.

Если один триггер применяется для нескольких *DML*-событий, то в условии триггера не будут применяться предикаты INSERTING, UPDATING и DELETING для идентификации инициирующего события. Рассмотрим пример использования предикатов триггера (предполагается, что в БД есть таблица *emp_log*):

```
CREATE OR REPLACE TRIGGER emp_log_t
AFTER INSERT OR UPDATE OR DELETE ON emp
FOR EACH ROW
DECLARE
dmltype CHAR(1);
```

```
BEGIN
IF INSERTING THEN dmltype := 'I';
INSERT INTO emp_log (emp_no, who, operation)
VALUES (:NEW.empno, USER, dmltype);
ELSIF UPDATING THEN dmltype:= 'U';
INSERT INTO emp_log (emp_no, who, operation)
VALUES (: new.empno, USER, dmltype);
END IF;
END;
```

Ошибки при изменении таблицы. Довольно часто при разработке триггеров на изменение данных в таблице приходится сталкиваться с проблемой «мутации» триггеров, а именно с ошибкой *Oracle*:

ORA-04091: *table … is mutating. trigger/function may not see it*

Эта ошибка возникнет, например, при разработке триггера на обновление таблицы сотрудников, где необходимо проводить проверку на величину заработной платы, чтобы она не превышала средней по отделу более чем на 20 %.

В общем, мутирующая таблица – это либо таблица, обновляемая в данный момент командой UPDATE, DELETE, INSERT, либо таблица, обновление которой может потребоваться вследствие срабатывания каскадного удаления в подчиненной таблице (ограничение DELETE CASCADE). Для триггеров, принадлежащих изменяемой таблице, эта таблица является мутирующей (к операторным триггерам это понятие не применяется). Кроме того, мутирующей таблицей является любая другая, ссылающаяся на таблицу триггера через ограничение внешнего ключа, что позволяет СУБД запретить чтение несогласованного набора данных.

```
CREATE OR REPLACE TRIGGER employees_sal
BEFORE UPDATE OF salary    ON employees
REFERENCING NEW AS NEW1
FOR EACH ROW
DECLARE
avr_sal NUMBER;
-- PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
SELECT AVG (salary) INTO avr_sal FROM employees
WHERE department_id =: NEW1.department_id;
-- COMMIT;
IF avr_sal < :NEW1.salary*1.2 THEN
RAISE_APPLICATION_ERROR (-20001,'заработная плата превышает среднюю по отделу на 20 %');
END IF;
END;
```

При попытке обновить заработную плату, например:

UPDATE employees SET salary = 9000 WHERE employee_id = 109;

будет получено сообщение об ошибке, однако есть несколько простых приемов, которые могут решить данную проблему, а именно:

1) если триггер выполняется как автономная транзакция, т. е. объявлен с инструкцией PRAGMA AUTONOMUS_TRANSACTION и в нем выполняется инструкция COMMIT, тогда в нем можно запрашивать содержимое таблицы, но не изменять его. Таким образом, убрав комментарии в примере, можно в данном случае решить проблему мутации;

2) триггер уровня строки не может считывать или записывать данные из таблицы, с которой он связан, но данное ограничение не касается триггеров уровня инструкции, что дает возможность провести необходимые действия.

В более сложных ситуациях, когда невозможно решить проблему вышеуказанными способами, используют или комбинацию переменных пакета с двумя триггерами, или COMPOUND-триггеры. В первом случае необходимо на уровне пакета объявить переменную типа ассоциативного массива, в которой можно будет глобально сохранить необходимые данные из мутирующей таблицы с помощью операторного триггера, а затем можно использовать данные, записанные в пакетной переменной с помощью строчного триггера.

Второй способ предполагает знание синтаксиса COMPOUND-триггера. Появившиеся в версии *Oracle*11*g* эти триггеры включают в одном блоке обработку всех видов *DML*-триггеров, могут содержать переменные, которые живут на всем протяжении выполнения оператора, вызвавшего срабатывание триггера, и включают следующие секции:

– BEFORE STATEMENT;
– AFTER STATEMENT;
– BEFORE EACH ROW;
– AFTER EACH ROW.

Некоторые правила написания:

– нет секции инициализации, но для этих целей можно использовать секцию *before statement*;

– нельзя обращаться к псевдозаписям :OLD, :NEW в секциях уровня оператора (*before statement* и *after statement*);

– изменять значения полей псевдозаписи :NEW можно только в секции *before each row*;

– исключения, сгенерированные в одной секции, нельзя обрабатывать в другой секции;

– если используется оператор *goto*, он должен указывать на код в той же секции.

Составной триггер можно использовать: для написания кода, соответствующего разным моментам времени события, и совместного использования переменных, объявленных в секции DECLARE; для сбора строк, предназначенных для загрузки в другую таблицу, в секции BEFOR/AFTER EACH ROW, которые затем вставляются в секции AFTER STATEMENT; для более простого решения проблемы мутирующей таблицы.

Общий синтаксис COMPOUND триггера:

CREATE OR REPLACE TRIGGER *имя_триггера*
FOR {INSERT|UPDATE|DELETE}[OF *имя_столбца*, …] ON *имя_таблицы*
COMPOUND TRIGGER
[*секция_объявления_переменных_триггера*]
BEFORE STATEMENT
секция_операторного_триггера;
BEFORE EACH ROW
секция_строчного_триггера;
AFTER EACH ROW
секция_строчного_триггера;
AFTER STATEMENT
секция_операторного_триггера;
END;

Например, с помощью COMPOUND-триггера будем контролировать, чтобы заработная плата работников не превышала допустимых значений минимума и максимума:

```
CREATE OR REPLACE TRIGGER sbiu_employees
FOR INSERT or UPDATE OF salary, job_id ON employees
COMPOUND TRIGGER
TYPE sal_rec_type IS RECORD (min_sal employees.salary%TYPE, max_sal employees.salary%TYPE);
TYPE sal_tab_type IS TABLE of sal_rec_type INDEX BY varchar2(30);
sal_tab sal_tab_type;

BEFORE STATEMENT IS
BEGIN
sal_tab.DELETE;
FOR rec IN (SELECT job_id, MIN(salary) min_sal, MAX(salary) max_sal
FROM employees GROUP BY job_id)
LOOP
sal_tab(rec.job_id).min_sal:= rec.min_sal;
sal_tab(rec.job_id).max_sal:= rec.max_sal;
END LOOP;
END BEFORE STATEMENT;
```

```
BEFORE EACH ROW IS
BEGIN
IF :NEW.salary NOT BETWEEN sal_tab (:NEW.job_id).min_sal AND
sal_tab(:NEW.job_id).max_sal
THEN RAISE_APPLICATION_ERROR (-20505, 'out of range');
END IF;
END BEFORE EACH ROW;
END;
```

***DDL*-триггеры.** Данные триггеры запускаются на события, изменяющие объекты БД: таблицы, индексы, триггеры. Синтаксис триггеров следующий:

```
CREATE [OR REPLACE] TRIGGER имя_триггера
{BEFOR | AFTER} {событие_DDL} ON {DATABASE | SCHEMA}
DECLARE
объявление_переменных;
BEGIN
код триггера;
END;
```

Список событий, на которые реагируют триггеры *DDL*, представлен в табл. 9.1.

Таблица 9.1

Список событий, на которые реагируют триггеры *DDL*

Событие	Описание
CREATE	Создание объекта
ALTER	Изменение объекта
DROP	Удаление объекта
ANALYZE	Сбор статистики по объекту
ASSOCIATE STATISTICS	Связывание статистики с объектом
AUDIT	Включение средств аудита
NOAUDIT	Отключение средств аудита
COMMENT	Создание комментария на объект
DDL	Наступление любого из *DDL*-событий
DIASSOCIATE STATISTICS	Удаление статистики
GRANT	Назначение прав пользователю
RENAME	Переименование объекта
REVOKE	Отмена прав пользователю
TRUNCATE	Очистка таблицы

Применение *DDL*-триггеров позволяет отслеживать, фиксировать, проверять, запрещать действия с объектами БД.

```
CREATE OR REPLACE TRIGGER drop_trigger
BEFORE DROP ON SCHEMA
BEGIN
RAISE_APPLICATION_ERROR (
num => -20000,
msg => 'Cannot drop object');
END;
```

Триггеры событий БД. Триггеры могут отслеживать и обрабатывать события экземпляра сервера БД. Триггер имеет следующий синтаксис:

```
CREATE OR REPLACE TRIGGER имя  триггера
{BEFOR | AFTER }
{SERVERERROR | LOGON | LOGOFF | STARTUP | SHUTDOWN | SUSPEND } ON { DATABASE | SCHEMA}
DECLARE
объявление_переменных;
BEGIN
код_триггера;
END;
```

Подробное описание событий и их определений представлено в табл. 9.2.

Таблица 9.2

События триггера уровня БД

Событие	Описание
SERVERERROR	Определяет триггер, срабатывающий в момент, когда *Oracle* генерирует какую-либо ошибку. Исключения составляют следующие сообщения: ORA-01403: *data not found*; ORA-01422: *exact fetch returns more than requested number of rows*; ORA-01423: *error encountered while checking for extra rows in exact fetch*; ORA-01034: *oracle not available*; ORA-04030: *out of process memory*
LOGON	Определяет триггер на событие LOGON – начало сеанса
LOGOFF	Определяет триггер на событие LOGOFF – закрытия сессии клиента
STARTUP	Определяет триггер на событие STARTUP – старт БД
SHUTDOWN	Определяет триггер на событие SHUTDOWN – закрытие БД
SUSPEND	Определяет триггер на событие «приостановление инструкции». Появился в версии *Oracle*9*i*.

Необходимо отметить, что системные триггеры ON DATABASE запускаются только в *SQL*Plus* из-под учетной записи администратора (*as sysdba*), например создадим таблицу *system.auditbase* и затем системный триггер:

```
CREATE TABLE system.auditbase
(nzap NUMBER,
polz VARCHAR2(20),
tmin TIMESTAMP,
oper VARCHAR2(50));

CREATE OR REPLACE TRIGGER fixuserm
AFTER LOGON ON DATABASE
BEGIN
INSERT INTO SYSTEM.AUDITBASE(nzap, polz, tmin, oper)
VALUES (1, USER, SYSDATE, 'UserIsLog(off)');
END fixuserm;
```

Триггеры INSTEAD OF. Создаются для представлений (*view*) и служат для замещения *DML*-операций своим функционалом; позволяют производить операции вставки/обновления или удаления для необновляемых представлений. Это всегда триггер уровня записи (*row level*), который имеет доступ к псевдозаписям :OLD и :NEW, но не может изменять их.

Например, создадим необновляемое представление:

```
CREATE OR REPLACE VIEW my_mgr_view AS
SELECT ( d.department_id || ' ' || d.department_name) "Department",
d.manager_id, e.first_name, e.last_name, e.email, e.hire_date "Hired On",
e.phone_number, e.salary, e.commission_pct,
(e.job_id || ' ' || j.job_title) "Job Class"
FROM departments d
JOIN employees e ON d.manager_id = e.employee_id
JOIN jobs j ON e.job_id = j.job_id
ORDER BY d.department_id;
```

Для того чтобы иметь возможность обновить представление, создадим триггер INSTEAD OF:

```
CREATE OR REPLACE TRIGGER update_my_mgr_view
INSTEAD OF UPDATE ON my_mgr_view
FOR EACH ROW
BEGIN
UPDATE employees
SET last_name = :NEW.last_name,
first_name = :NEW.first_name,
email = :NEW.email,
```

```
phone_number = :NEW.phone_number,
salary = :NEW.salary,
commission_pct = :NEW.commission_pct
WHERE employee_id = :OLD.manager_id;
END;
```

Теперь попробуем обновить его, т. е. выполним следующую команду:

```
UPDATE my_mgr_view
SET first_name = 'Denis' WHERE manager_id = 114;
```

Последовательность событий при срабатывании триггеров. Для одной таблицы может быть определено множество различных типов триггеров. Необходимо понимать, в каком порядке они срабатывают. Основное влияние на последовательность выполнения оказывает конструкция FOR EACH ROW.

К *DML*-событиям относятся команды INSERT, UPDATE и DELETE для таблицы или представления. Для этих событий могут быть определены триггеры уровня команды (только для таблиц) или уровня строки и запускаться они могут как до, так и после инициирующего события. Триггер BEFORE может изменять данные в обрабатываемых строках и обычно применяется для того, чтобы определить, должна ли выполняться инициировавшая его команда. Триггер AFTER не выполняет это дополнительное логическое чтение, поэтому выполняется немного быстрее, но зато они не могут изменять новые значения.

Триггеры DML, если они определены, запускаются в следующем порядке:

1) триггер BEFORE уровня команды, для каждой строки, обработанной командой;
2) триггер BEFORE уровня строки;
3) инициирующая команда;
4) триггер AFTER уровня строки;
5) триггер AFTER уровня команды.

К событиям *DDL* относят CREATE, ALTER, DROP. *DDL*-триггеры могут применяться к какой-то одной схеме или же ко всей БД.

К событиям экземпляра сервера БД относятся SERVERERROR, LOGON, LOGOFF, STARTUP, SHUTDDOWN. Триггер SHUTDOWN срабатывает в случае события SHUTDDOWN NORMAL и SHUTDOWN IMMEDIATE, но не SHUTDOWN ABORT.

Контрольные вопросы и задания к разделу 9

1. Назовите общие ключевые слова процедур и функций, поясните назначение каждого из них.
2. Какие способы вызова функций могут применяться?

3. Какие требования предъявляются к вызову хранимых функций в *SQL*?

4. Что понимается под термином «табличная функция»?

5. Какие способы перегрузки программ известны?

6. Для чего используются локальные программы?

7. Создайте и выполните процедуру по изменению заработной платы указанному сотруднику из подразд. 9.2.

8. Измените код процедуры таким образом, чтобы заработная плата изменялась у всех сотрудников заданного отдела.

9. Создайте и выполните различными способами функцию, которая возвращает количество отработанных сотрудником лет и месяцев.

10. Измените код функции из предыдущего задания таким образом, чтобы она возвращала количество лет и месяцев, которые сотрудник не работал с даты своего рождения.

11. Создайте и выполните табличную функцию из подразд. 9.4.

12. Создайте и выполните пример с локальными модулями из подразд. 9.4.

13. Измените код процедуры из предыдущего задания таким образом, чтобы в нем были перегруженные локальные модули, позволяющие изменять заработную плату не только по коду, но также по имени и фамилии сотрудника.

14. Выполните все примеры из подразд. 9.6.

10. СТАНДАРТНЫЕ ПАКЕТЫ

Стандартные пакеты СУБД *Oracle* – это мощный инструмент разработчика, потому что гораздо проще разрабатывать и сопровождать код, использующий стандартные средства, чем создавать их самостоятельно. Корпорация *Oracle* поставляет пакеты для определенных целей (например, шифрования данных, управления почтой и т. п.) и их существует достаточно большое количество. Пакет с именем STANDARD определяет саму среду *PL/SQL*. Спецификация пакета глобально объявляет типы, исключения и подпрограммы, которые автоматически доступны для всех программ *PL/SQL* без указания полного имени пакета, хотя имена всех стандартных пакетов *Oracle* начинаются с префиксов DBMS_ (*database management system*) – это пакеты для работы с БД или UTL_(*universal tag locator*) – пакеты общего назначения. Большинство пакетов хранится в БД в скомпилированном, скрытом (*wrapped*) формате. Обратившись к представлению словаря данных ALL_SOURCE, можно увидеть спецификацию пакета, но не его реализацию.

10.1. Пакет DBMS_SCHEDULER

Хорошей альтернативой триггерам, позволяющей автоматизировать выполнение задания внутри БД *Oracle*, является планировщик (SCHEDULER) – специальная программа, которая запускается в определенное время. В СУБД *Oracle* он реализован с помощью стандартного пакета DBMS_SCHEDULER. Пакет позволяет запускать хранимую процедуру или анонимный блок *PL/SQL* в моменты времени, вычисляемые по указанной пользователем формуле. Он использует следующие основные понятия:

– расписание (*schedule*);
– программа (*program*);
– плановое задание (*job*).

Планировщик можно настроить на однократный запуск (непосредственно после запуска или в определенное время) и на многократный запуск (запуск по расписанию).

Прежде всего для написания планировщика нужно, как правило, дать права на его создание командой:

GRANT CREATE JOB TO *имя_пользователя*;

Рассмотрим пример планировщика для однократного запуска анонимного блока:

```
BEGIN
DBMS_SCHEDULER.CREATE_JOB
(job_name => 'simple_job',
job_type => 'PLSQL_BLOCK',
```

```
job_action => 'UPDATE employees SET salary=salary+1;',
enabled => TRUE);
END;
```

где *job_name* – имя планировщика; *job_type* – тип запускаемого задания, который может принимать следующие значения: 'STORED_PROCEDURE' – хранимая процедура, 'PLSQL_BLOCK' – анонимный блок; *job_action* – то, что запускается; *enabled* – определяет, включен или выключен планировщик.

В рассмотренном примере заработная плата каждому сотруднику увеличится на один, и планировщик удалится автоматически, что равносильно выполнению обычной команды UPDATE.

Для запуска планировщика однократно в определенное время дополнительно может использоваться параметр пакета *start_date*, например:

```
start_date => SYSTIMESTAMP + INTERVAL '10' SECOND
```

где *start_date* – это время запуска, которое в данном примере составляет 10 с от текущего времени.

Для многократного запуска может быть введен параметр пакета *repeat_interval*, пример использования которого представлен ниже:

```
repeat_interval => 'FREQ = MONTHLY; BYDAY = SUN, -1 SAT'
```

В данном примере задание будет исполняться ежемесячно по воскресеньям и последним субботам месяца, т. е. в планировщике есть возможность указывать частоту, интервал и спецификатор запуска задания, например могут быть следующие варианты:

– FREQ = HOURLY; INTERVAL = 4 – запуск каждые 4 часа;

– FREQ = HOURLY; INTERVAL = 4; BYMINUTE = 10; BYSECOND = 30 – запуск каждые 4 часа на 10-й минуте, 30-й секунде;

– FREQ = YEARLY; BYYEARDAY = 276 – запуск каждый 276-й день года;

– FREQ = YEARLY; BYMONTH = MAR; BYMONTHDAY = 31 – запуск каждое 31-е марта.

Частота – это обязательный компонент календарного выражения, который идентифицируется ключевым словом FREQ. Возможные значения – YEARLY, MONTHLY, DAILY, HOURLY, MINUTELY и SECONDLY.

Интервал повторения идентифицируется ключевым словом INTERVAL и указывает, насколько часто БД должна повторять задание.

Спецификаторы предоставляют детальную информацию о том, когда должно запускаться задание. Возможные значения – BYMONTH, BYWEEKNO, BYYEARDAY, BYMONTHDAY, BYDAY, BYHOUR, BYMINUTE и BYSECOND. Спецификаторы являются необязательными.

В анонимном блоке можно использовать команды:

```
-- отключить планировщик
DBMS_SCHEDULER.DISABLE ('simple_job');

-- остановить выполняющиеся задания
DBMS_SCHEDULER.STOP_JOB ('simple_job');

-- запустить задания
DBMS_SCHEDULER.RUN_JOB ('simple_job');

-- запустить задания в фоновом режиме
DBMS_SCHEDULER.RUN_JOB ('simple_job',
use_current_session => false);

-- удалить задания
DBMS_SCHEDULER.DROP_JOB ('simple_job');
```

или

```
DBMS_SCHEDULER.DROP_JOB ('simple_job', force => TRUE);
```

Также можно изменить параметры планировщика, выполняя последовательно приведенные далее действия:

```
--отключить планировщик

BEGIN
DBMS_SCHEDULER.DISABLE ('simple_job', TRUE);
END;

-- изменить атрибуты
BEGIN
DBMS_SCHEDULER.SET_ATTRIBUTE (
name => 'simple_job',
attribute => 'repeat_interval',
value => 'FREQ = YEARLY; BYMONTH = MAR; BYMONTHDAY = 31');
DBMS_SCHEDULER.SET_ATTRIBUTE (
name => 'simple_job',
attribute => 'STOP_ON_WINDOW_CLOSE', value => TRUE);
END;

-- включить планировщик
BEGIN
DBMS_SCHEDULER.ENABLE ('simple_job');
END;
```

Информацию об уже созданных планировщиках можно просмотреть в представлении словаря данных USER_SCHEDULER_JOBS.

Кроме того, DBMS_SCHEDULER позволяет скомпоновать задание из независимых элементов: программы и расписания. Оба эти элемента самостоятельны и их можно комбинировать в разных заданиях и изменять, не внося изменений в определения заданий.

Пример создания программы:

```
BEGIN
DBMS_SCHEDULER.CREATE_PROGRAM
(program_name   => 'simple_program',
program_type     => 'STORED_PROCEDURE',
program_action   => 'updatesal',
enabled             => TRUE);
END;
```

Список сведений об имеющихся программах для планировщика указан в таблице словаря USER_SCHEDULER_PROGRAMS.

Другими значениями параметра PROGRAM_TYPE могут быть 'PLSQL_BLOCK' и 'EXECUTABLE'.

При наличии у процедуры параметров их количество требуется указать:

```
CREATE PROCEDURE salary1 (deer NUMBER ) AS
BEGIN
UPDATE employees SET salary = salary – deer;
END;
```

```
BEGIN
DBMS_SCHEDULER.CREATE_PROGRAM
( program_name     => 'simple_program1',
 program_type       => 'STORED_PROCEDURE',
 program_action     => 'salary1',
 enabled              => FALSE,
 number_of_arguments => 1);
END;
```

Необходимо обратить внимание, что программа создана «отключенной», потому что указать фактические значения параметров во «включенной» программе нельзя, так что в дальнейшем последовательность действий будет следующая:

```
BEGIN
DBMS_SCHEDULER.DEFINE_PROGRAM_ARGUMENT
( program_name      => ' ,
 argument_position => 1,
 argument_name     => 'DELTA',
 argument_type      => 'NUMBER');
END;
```

```
BEGIN
DBMS_SCHEDULER.ENABLE ('simple_program1');
END;
```

Пример создания расписания:

```
BEGIN
DBMS_SCHEDULER.CREATE_SCHEDULE
(schedule_name => 'simple_schedule',
start_date => SYSTIMESTAMP,
repeat_interval => 'FREQ = WEEKLY; BYDAY = MON, TUE, WED, THU,
FRI',
end_date => SYSTIMESTAMP + INTERVAL '1' MONTH);
END;
```

Список сведений об имеющихся расписаниях для планировщика находится в таблице словаря USER_SCHEDULER_SCHEDULES.

В общем случае синтаксис указания графика для расписания (параметр REPEAT_INTERVAL) допускает ссылаться на ранее созданные таким же образом расписания.

Из самостоятельно существующих программы и расписания можно составить скомпонованное задание, например:

```
BEGIN
DBMS_SCHEDULER.CREATE_JOB ( job_name => 'compound_job',
program_name  => 'simple_program',
schedule_name => 'simple_schedule',
enabled =>TRUE);
END;
```

10.2. Динамический *SQL*. Понятие привязки

Из всех *SQL*-операторов в программах *PL/SQL* напрямую можно использовать лишь операторы *DML* и операторы управления транзакциями. Операторы *DDL* так использовать нельзя. Чтобы пояснить смысл этих ограничений, рассмотрим принцип создания программ *PL/SQL*.

В любом языке программирования привязка переменных может быть либо ранней, либо поздней. *Привязка* (*binding*) переменной – это процесс указания области памяти, соответствующей некоторому идентификатору программы. В *PL/SQL* в процесс привязки входит также проверка БД на наличие полномочий, позволяющих обращаться к объектам схем. В том языке, где используется *ранняя привязка* (*early binding*), этот процесс осуществляется на этапе компиляции программы, а в языке, где применяется *поздняя привязка* (*late binding*), – откладывается до времени выполнения программы. Ранняя привязка означает, что этап компиляции программы будет занимать большее время (так как при этом нужно

привязывать переменные), однако выполняться программа будет быстрее, поскольку к этому моменту привязка будет завершена. Таким образом, поздняя привязка сокращает время компиляции, но увеличивает время выполнения программы.

При разработке *PL/SQL* было принято решение об использовании ранней привязки, чтобы объекты БД к моменту выполнения блока были уже проверены и чтобы блок мог быть выполнен максимально быстро. Это вполне оправданно, так как блоки *PL/SQL* можно хранить в БД как процедуры, функции, пакеты и триггеры. Такие объекты хранятся в скомпилированном виде, т. е. при необходимости их можно загрузить из БД и выполнить.

Кроме этого, операторы *DDL* модифицируют объекты БД, следовательно, полномочия на объект должны быть вновь подтверждены, а процесс подтверждения полномочий также требует привязки идентификаторов.

Рассмотрим блок, приведенный в источнике [8]:

```
DECLARE
v_SQLString VARCHAR2(100);
v_SetClause VARCHAR 2(100);
v_WhereClause VARCHAR2(100);
BEGIN
v_SetClause := 'SET salary = :salary WHERE';
v_WhereClause := 'position = :v_position';
v_SQLString := 'UPDATE staff' || v_SetClause || v_WhereClause
DoIt(v_SQLString);
END;
```

Этот блок нельзя запустить на выполнение сразу же после его написания, так как предварительно нужно создать процедуру **DoIt()**.

Отметим, что посредством создания *корректной функции* **DoIt()** всегда можно выполнить *произвольную SQL*-инструкцию, задав набор подстрок этой инструкции, а затем осуществив их конкатенацию (||).

Однако в таком случае функция **DoIt()** должна осуществлять:

– проверку корректности задания *SQL*-инструкции, т. е. выполнять грамматический разбор строки инструкции с целью выявления синтаксических и семантических ошибок;

– связывание параметров *SQL*-инструкции;

– проверку корректного выполнения *Oracle Server* заданной инструкции.

Таким образом, реализовать функцию, обрабатывающую любую *SQL*-инструкцию, практически очень сложно из-за произвольности количества и типов связываемых параметров различных инструкций. Однако создать функцию наподобие **DoIt()**, которая выполняла бы инструкции с фиксированным количеством параметров определенного типа, в принципе возможно.

Для реализации подобных функций используется **динамический *SQL***. Для динамических операторов устанавливается меньшее число ограничений: многое

из того, что необходимо знать во время компиляции, вовсе необязательно описывать до этапа выполнения программы. Полное описание *SQL*-оператора можно дать во время его выполнения.

Итак, если структура *SQL*-операторов известна уже во время компиляции, то программы являются *статическими*. Однако если привязку параметров и определение прав доступа нужно определять «на лету», то речь идет о динамическом *SQL*, который обеспечивает возможность использования всех *SQL*-операторов в *PL/SQL*.

В *Oracle* существует два способа применения динамического *SQL*:

1) Стандартный пакет *PL/SQL* – DBMS_SQL.

2) Встроенный динамический *SQL*, который появился в *Oracle*8*i*. При его использовании динамические операторы управления встраиваются непосредственно в язык *PL/SQL* и поэтому выполняются значительно быстрее, чем в пакете DBMS_SQL.

Пакет DBMS_SQL используется, если на этапе компиляции неизвестна структура курсора, число и типы переменных привязки. Он позволяет выяснить структуру курсора в реальном режиме времени, но имеет достаточно сложный синтаксис.

Встроенный динамический *SQL* (*Native Dynamic SQL – NDS*) основывается на применении оператора EXECUTE IMMEDIATE и курсорном цикле OPEN … FOR, которые имеют простой и короткий синтаксис, но при этом число и типы переменных привязки должны быть известны во время компиляции, как и структура курсора.

10.3. Пакет DBMS_SQL

Общий алгоритм выполнения *SQL*-операторов с помощью DBMS_SQL состоит из следующих этапов:

– преобразование *SQL*-оператора в строку символов – синтаксический разбор;

– грамматический разбор строки символов с помощью DBMS_SQL.PARSE;

– привязка всех входных переменных с помощью DBMS_SQL.BIND_VARIABLE;

– если выполняемый оператор – это оператор DML (UPDATE, DELETE, INSERT) или DDL, то выполнение его с помощью DBMS_SQL.EXECUTE с последующим считыванием выходных переменных привязки посредством DBMS_SQL.VARIABLE_VALUE (если нужно);

– если оператор осуществляет выборку данных (SELECT) – сначала описание выходных переменных с помощью DBMS_SQL.DEFINE_COLUMN;

– затем выполнение запроса на выборку с помощью DBMS_SQL.EXECUTE и выборка результатов при помощи DBMS_SQL.FETCH_ROWS и DBMS_SQL.COLUMN_VALUE.

Рассмотрим пример: процедура сначала удаляет, а затем повторно создает временную таблицу; описание таблицы передается процедуре как аргумент.

```
CREATE OR REPLACE PROCEDURE RecreateTempTable (p_Description IN VARCHAR2) IS
v_Cursor NUMBER;
v_CreateString VARCHAR2(100);
v_DropString VARCHAR2(100);

BEGIN
--открываем курсор для обработки данных;
v_Cursor:=DBMS_SQL.OPEN_CURSOR;
/* удаляем таблицу, для чего проводится грамматический разбор команды 'DROP TABLE' и затем ее выполнение. Если таблица не существует, то будет вызвана ошибка ORA-942 */
v_DropString:='DROP TABLE temptable';
BEGIN
DBMS_SQL.PARSE (v_Cursor, v_DropString, DBMS_SQL.NATIVE);
EXCEPTION
WHEN OTHERS THEN
IF SQLCODE != -942 THEN
RAISE;
END IF;
END;

/* Создаем таблицу. Сначала необходимо инициализировать строку символов CREATE TABLE, а затем провести грамматический разбор и выполнить DDL-оператор*/
v_CreateString:= 'CREATE TABLE temptable' || p_Description;
DBMS_SQL.PARSE (v_Cursor, v_CreateString, DBMS_SQL.NATIVE);
--закрываем курсор;
DBMS_SQL.CLOSE_CURSOR (v_Cursor);

EXCEPTION
WHEN OTHERS THEN
/* сначала закрываем курсор, а затем переустанавливаем ошибку, чтобы передать ее в вызывающий блок*/
DBMS_SQL.CLOSE_CURSOR (v_Cursor);
RAISE;
END RecreateTempTable;
```

При анализе рассмотренного примера необходимо обратить внимание на следующее:

– строка символов, грамматический разбор которой выполняется, может быть константой (*v_DropString*), а также может создаваться программой динамически, с помощью символьных функций, например с помощью функции конкатенации (*v_CreateString*);

– обработка ошибок осуществляется так же, как в статическом *SQL*, ошибки устанавливаются и обрабатываются с помощью исключительных ситуаций. Отличие в том, что теперь во время выполнения блока могут возвращаться и ошибки компиляции;

– пользователю предоставляются большие возможности по управлению курсорами, в частности определение моментов их открытия и закрытия. Все открытые курсоры должны быть закрыты даже при установлении исключительной ситуации. В данном случае закрытие курсора обеспечивается вызовом DBMS_SQL.CLOSE_CURSOR в обработчике исключительной ситуации.

Выполняется динамическая процедура так же, как и статическая:

```
BEGIN
RecreateTempTable('(num_col NUMBER, char_col VARCHAR2(200))');
END;
```

На рис. 10.1 приведена схема алгоритма использования пакета DBMS_SQL.

С помощью DBMS_SQL можно обрабатывать операторы трех видов: *SQL*-операторы (*DML, DDL* и ALTER SESSION), запросы и анонимные блоки *PL/SQL*. Каждый вид обрабатывается при помощи процедур, которые описаны ниже:

– OPEN_CURSOR. Как в статическом *SQL*, так и в динамическом *PL/SQL* каждый *SQL*-оператор выполняется в границах курсора. Кроме того, в динамическом *PL/SQL* можно управлять процессом обработки курсора. OPEN_CURSOR возвращает идентификационный номер курсора, используемый для обозначения контекстной области, в которой будет выполняться оператор. Этот номер будет указываться во всех последующих вызовах данного курсора.

– PARSE. Грамматический разбор оператора заключается в посылке его на сервер, где проверяются синтаксис и семантика данного оператора. Если он является запросом, то на этом этапе определяется план его выполнения. Если это оператор *DDL*, то он также выполняется.

– BIND_VARIABLE. Привязка переменной к заполнителю аналогична процессу привязки, используемому в *PL/SQL* для статических *SQL*-операторов. Заполнитель (*placeholder*) – это специальный идентификатор в последовательности символов оператора. В процессе привязки устанавливается соответствие между заполнителем и фактической переменной, а пакету DBMS_SQL сообщается о типе и размере переменной. Привязка выполняется для входных переменных в *SQL*-операторах и для входных и выходных переменных в операторах *PL/SQL*. Переменные, использующиеся таким образом, называют *переменными привязки.*

– DEFINE_COLUMN. Описание элемента списка выбора аналогично привязке входной переменной. В данном случае переменными являются результаты запроса. Процедура DEFINE_COLUMN определяет тип и размер переменных PL/SQL, в которые будут записываться данные при считывании информации процедурой FETCH_ROWS. DEFINE_COLUMN используется только для элементов списка выбора (результатов запросов SELECT), а BIND_VARIABLE – для входных или выходных переменных (заполнителей, указанных в операторах).

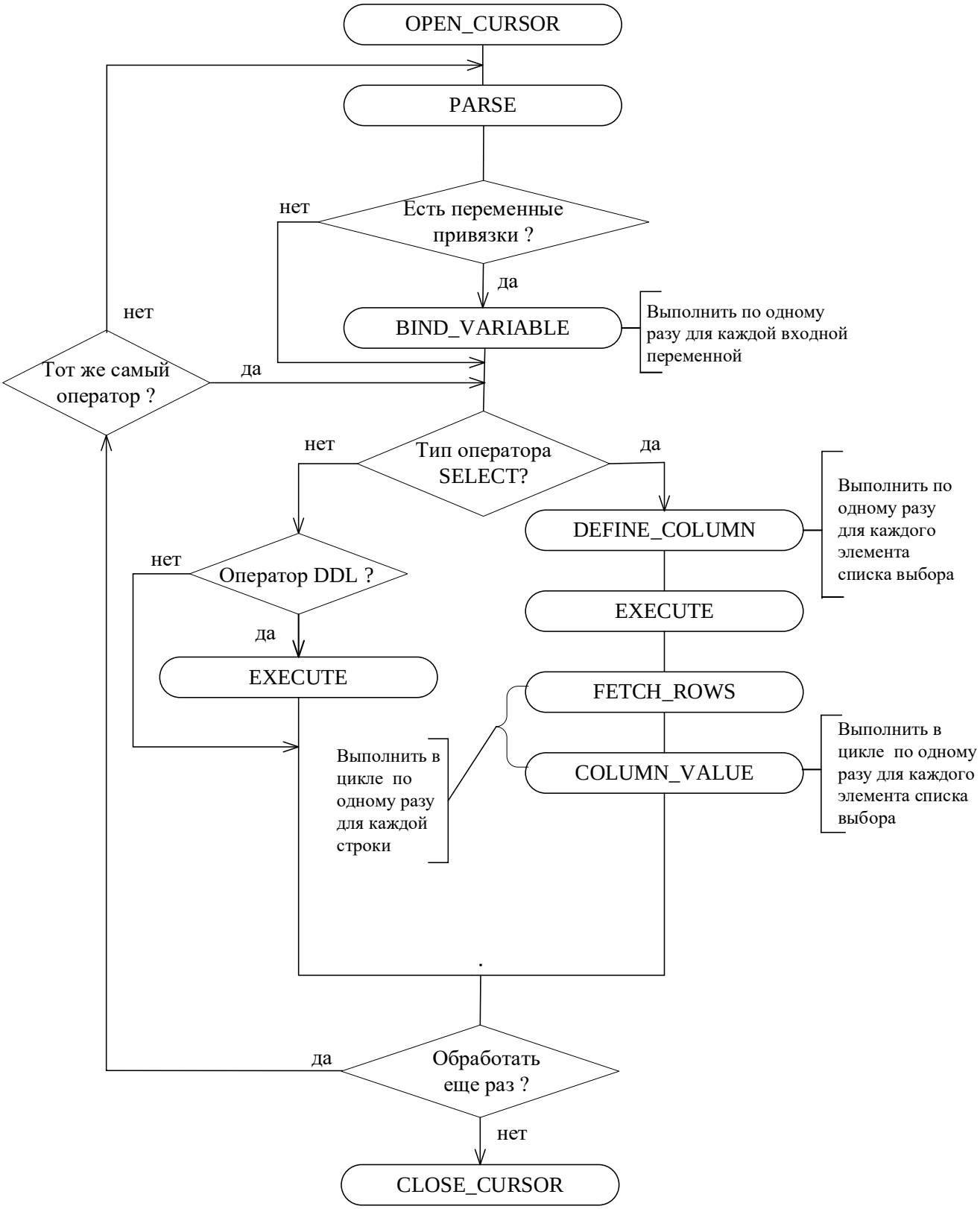


Рис. 10.1. Схема алгоритма обработки данных в DBMS_SQL

– EXECUTE. Исполняет операторы, не являющиеся запросами (SELECT), и возвращает число обработанных строк. Для запросов EXECUTE определяет активный набор (набор строк, удовлетворяющих условию запроса WHERE). После этого данные считываются процедурой FETCH_ROWS. Для любых операторов переменные привязки анализируются во время работы функции EXECUTE.

– FETCH_ROWS. С каждым вызовом процедуры FETCH_ROWS с сервера считывается все больший объем данных. Затем полученная информация преобразуется в типы данных, указанных процедурой DEFINE_COLUMN. Процедура EXECUTE_AND_FETCH позволяет объединить операции обработки и считывания данных в одном вызове.

– COLUMN_VALUE. Возвращает фактическую информацию после вызова FETCH_ROWS. В ней используются переменные того типа, который указан процедурой DEFINE_COLUMN. Следует применять только для запросов.

– CLOSE_CURSOR. После окончания обработки данных курсор закрывается. При этом ресурсы, использовавшиеся курсором, освобождаются.

10.4. Обработка операторов DML в DBMS_SQL

Для обработки операторов UPDATE, DELETE, INSERT средствами пакета DBMS_SQL необходимо последовательно выполнить следующие действия:

1. Открыть курсор. Осуществляется посредством вызова процедуры OPEN_CURSOR, описание которой в пакете выглядит следующим образом:

```
OPEN_CURSOR RETURN INTEGER;
```

Параметры в данной процедуре отсутствуют.

Каждый вызов возвращает целое число, представляющее собой идентификационный номер курсора. Этот номер используется в последующих вызовах курсора. В границах одного курсора можно по очереди обрабатывать несколько *SQL*-операторов или выполнять один и тот же оператор несколько раз.

2. Выполнить грамматический разбор оператора. При выполнении грамматического разбора оператор направляется на сервер БД. Сервер проверяет его синтаксис и семантику и возвращает ошибку (устанавливая исключительную ситуацию), если нарушены требования грамматики. Кроме того, во время разбора определяется план выполнения оператора. Осуществляется грамматический разбор посредством вызова процедуры DBMS_SQL.PARSE, описание которой в пакете имеет следующий вид:

```
PROCEDURE PARSE (c IN INTEGER,
statement IN VARCHAR2,
language_flag IN INTEGER);
```

где *c* – идентификационный номер курсора, предварительно должен быть открыт посредством OPEN_CURSOR; *statement* – оператор, грамматический разбор ко-

торого выполняется; *language_flag* – указывает, как трактовать оператор, его значение NATIVE указывает на режим использования тех наборов символов, которые установлены для БД, с которой выполнено соединение.

3. Привязка входных переменных. При выполнении операции указанные в нем заполнители связываются с фактическими переменными. Имена заполнителей обычно предваряют символом двоеточия. Процедура BIND_VARIABLE выполняет привязку и объявление имен заполнителей. Размер и тип данных фактических переменных также устанавливается BIND_VARIABLE посредством набора переопределенных вызовов:

PROCEDURE BIND_VARIABLE (*c* IN INTEGER, *name* IN VARCHAR2, *value* IN NUMBER)
PROCEDURE BIND_VARIABLE (*c* IN INTEGER, *name* IN VARCHAR2, *value* IN VARCHAR2)
PROCEDURE BIND_VARIABLE (*c* IN INTEGER, *name* IN VARCHAR2, *value* IN VARCHAR2, *out_value_size* IN INTEGER)

где *name* – это имя заполнителя, с которым будет связана переменная; *value* – реальные данные, которые будут привязываться, тип и размер этой переменной также считываются (при необходимости данные, содержащиеся в этой переменной, будут преобразованы); *out_value_size* – это параметр, задаваемый при привязке переменных *varchar2* и *char*; если он указан, то это максимальный ожидаемый размер значения в байтах, если не указан, то используется размер, указанный в параметре *value*.

4. Выполнение оператора. Осуществляется посредством функции EXECUTE. Описание ее в пакете выглядит следующим образом:

FUNCTION EXECUTE (*c* IN INTEGER) RETURN INTEGER;

где *c* – идентификатор предварительно открытого курсора.

Функция EXECUTE возвращает число отработанных строк (в этом смысле возвращаемое значение аналогично курсорному атрибуту %ROWCOUNT). Следует учесть, что возвращаемое значение не определено для операторов выборки, а также и то, что EXECUTE вызывается из выражений программ.

5. Закрытие курсора. Закрытие курсора осуществляется посредством вызова процедуры CLOSE_CURSOR, которая выглядит следующим образом:

PROCEDURE CLOSE_CURSOR (*c* IN OUT INTEGER);

Передаваемое процедуре значение должно быть достоверным идентификатором курсора. После вызова фактический параметр устанавливается в NULL, что свидетельствует о закрытии курсора.

Приведем пример простой динамической процедуры, изменяющей адрес сотрудника:

```
CREATE OR REPLACE PROCEDURE update_address
(p_lname IN staff.lname%TYPE, p_fname IN staff.fname%TYPE, p_newaddress IN staff.address%TYPE, p_rowsupdated OUT INTEGER) IS
v_cursor_id INTEGER;
v_updatestmt VARCHAR2(100);
BEGIN
v_cursor_id := DBMS_SQL.OPEN_CURSOR;
v_updatestmt := 'UPDATE staff
                 SET address =:na
                  WHERE fname =: fname AND lname =: lname';
DBMS_SQL.PARSE (v_cursor_id, v_updatestmt, DBMS_SQL.NATIVE);
DBMS_SQL.BIND_VARIABLE (v_cursor_id, ':na', p_newaddress);
DBMS_SQL.BIND_VARIABLE (v_cursor_id, ':fname', p_fname);
DBMS_SQL.BIND_VARIABLE (v_cursor_id, ':lname', p_lname);
p_rowsupdated:=DBMS_SQL.EXECUTE (v_cursor_id);
DBMS_SQL.CLOSE_CURSOR(v_cursor_id);
EXCEPTION
WHEN OTHERS THEN
DBMS_SQL.CLOSE_CURSOR(v_cursor_id);
RAISE;
END update_address;
```

Выполним процедуру:

```
DECLARE
col INTEGER;
BEGIN
update_address('Виктор', 'Якубов', 'Якубова 482', col);
DBMS_OUTPUT.PUT_LINE (col);
END;
```

10.5. Обработка запросов на извлечение информации в DBMS_SQL

Обработка запросов на извлечение информации происходит в следующей последовательности [8]:

1) открыть курсор (OPEN_CURSOR);
2) выполнить грамматический разбор (PARSE);
3) выполнить привязку входных переменных (BIND_VARIABLE);
4) описать элементы списка выбора (DEFINE_COLUMN);
5) выполнить запрос (EXECUTE);
6) считать строки (FETCH_ROWS);
7) записать результаты в переменные (COLUMN_VALUE);
8) закрыть курсор (CLOSE_CURSOR).

Для динамического выполнения запросов необходимо использовать те же процедуры пакета DBMS_SQL, что и при выполнении инструкций *DML*, за исключением DEFINE_COLUMN, FETCH_ROWS и COLUMN_VALUE. Рассмотрим синтаксис этих процедур.

Процедура DEFINE_COLUMN определяет элементы списка выбора аналогично привязке входных переменных, за исключением того, что элементы списка выбора должны быть не привязаны, а только определены. В процедуре DEFINE_COLUMN указываются типы и размер переменных, в которые считываются элементы списка выбора. Каждый элемент при этом преобразуется в тип соответствующей переменной.

PROCEDURE DEFINE_COLUMN (*c* IN INTEGER, *position* IN INTEGER, *column* IN NUMBER)

PROCEDURE DEFINE_COLUMN (*c* IN INTEGER, *position* IN INTEGER, *column* IN VARCHAR2, *column_size* IN INTEGER)

Для переменных типа VARCHAR2 нужно обязательно указывать параметр *column_size*, поскольку система поддержки *PL/SQL* должна знать максимальный размер этих переменных во время выполнения программы, так как в отличие от NUMBER, DATE данные этих типов не имеют фиксированной длины, заранее известной компилятору. Тип и предназначение параметров процедуры DEFINE_COLUMN приведены в табл. 10.1.

Таблица 10.1

Параметры процедуры DEFINE_COLUMN

Параметр	Тип	Предназначение
c	INTEGER	Идентификатор курсора
position	INTEGER	Позиция пункта списка выбора
column	NUMBER, VARCHAR2	Переменная, определяющая тип и размер выходной переменной. Имя переменной не играет особой роли, однако тип и размер важны. По факту в DEFINE_COLUMN и COLUMN_VALUE обычно используются одни и те же переменные
column_size	INTEGER	Максимальный ожидаемый размер данных. Обязателен для тех типов, длина которых не известна заранее системе поддержки *PL/SQL*

Функция FETCH_ROWS считывает строки набора запроса в буфер. Синтаксис процедуры:

FUNCTION FETCH_ROWS (*c* IN INTEGER) RETURN INTEGER;

Функция возвращает число считываемых строк. FETCH_ROWS и COLUMN_VALUE вызываются в цикле несколько раз до тех пор, пока FETCH_ROWS не возвратит NULL.

Процедура COLUMN_VALUE используется для записи результатов в переменные *PL/SQL*. Эта процедура вызывается после успешного выполнения FETCH_ROWS. Если в выборке не были возвращены строки, COLUMN_VALUE устанавливает для выходной переменной NULL-значение. Ниже приведено описание процедуры в пакете DBMS_SQL:

PROCEDURE COLUMN_VALUE (*c* IN INTEGER, *position* IN INTEGER, *value* OUT NUMBER);

PROCEDURE COLUMN_VALUE (*c* IN *INTEGER*, *position* IN INTEGER, *value* OUT NUMBER, *column_error* OUT NUMBER, *actual_length* OUT NUMBER);

PROCEDURE COLUMN_VALUE (*c* IN INTEGER, *position* IN INTEGER, *value* OUT VARCHAR2);

PROCEDURE COLUMN_VALUE (*c* IN INTEGER, *position* IN INTEGER, *value* OUT VARCHAR2, *column_error* OUT NUMBER, *actual_length* OUT NUMBER).

Тип и назначение параметров процедуры COLUMN_VALUE приведены в табл. 10.2.

Таблица 10.2

Параметры процедуры COLUMN_VALUE

Параметр	Тип	Предназначение
c	INTEGER	Идентификатор курсора
position	INTEGER	Относительная позиция в списке выбора, как и в DEFINE_COLUMN, позиция первого элемента списка = 1
value	NUMBER, VARCHAR2	Выходная переменная, если тип этого параметра отличается от типа, указанного в DEFINE_COLUMN, то возникает ошибка, что соответствует исключительной ситуации DBMS_SQL.INCONSISTENT_TYPES
column_error	NUMBER	Код ошибки столбца выдается в виде отрицательного числа. Ошибка будет устанавливать исключительную ситуацию, а *column_error* позволяет определить, какой из столбцов стал причиной конкретной ошибки. Если столбец был успешно прочитан, то *column_error* = 0
actual_length	NUMBER	Если указан, то в данной переменной будет находиться исходный размер столбца (размер столбца перед его считыванием). Это удобно в случае, когда размер переменной недостаточен и значение усекается (это также приводит к ошибке)

Рассмотрим пример [9]. Процедура определяет имена и фамилии сотрудников по заданной должности:

```
CREATE OR REPLACE PROCEDURE DynamicQuery
(p_position IN staff.position %TYPE) IS
v_cursor_id INTEGER;
v_select_stmt VARCHAR2(500);
v_first_name staff.fname %TYPE;
v_last_name staff.lname %TYPE;
v_dummy INTEGER;
BEGIN
v_cursor_id := DBMS_SQL.OPEN_CURSOR;
v_select_stmt := 'SELECT lname, fname FROM staff
                  WHERE position = :pos
                  ORDER BY lname';
DBMS_SQL.PARSE (v_cursor_id, v_select_stmt, DBMS_SQL.NATIVE);
DBMS_SQL.BIND_VARIABLE (v_cursor_id, ':pos', p_position);
DBMS_SQL.DEFINE_COLUMN (v_cursor_id, 1, v_last_name, 25);
DBMS_SQL.DEFINE_COLUMN (v_cursor_id, 2, v_first_name, 25);
v_dummy:= DBMS_SQL.EXECUTE (v_cursor_id);
LOOP
IF DBMS_SQL.FETCH_ROWS (v_cursor_id)=0 THEN
EXIT;
END IF;
DBMS_SQL.COLUMN_VALUE (v_cursor_id, 1, v_last_name);
DBMS_SQL.COLUMN_VALUE (v_cursor_id, 2, v_first_name);
INSERT INTO temp_table (name_col)
VALUES (v_last_name || ' ' || v_first_name);
END LOOP;
DBMS_SQL.CLOSE_CURSOR (v_cursor_id);
COMMIT;
EXCEPTION
WHEN OTHERS THEN
DBMS_SQL.CLOSE_CURSOR (v_cursor_id);
RAISE;
END DynamicQuery;
```

Выполним процедуру:

```
BEGIN
DynamicQuery ('manager');
END;
```

10.6. Встроенный динамический *SQL*

Пакет *NDS* привносит в программы *PL/SQL* три новые функциональности:
– динамические *DDL* и *DML* без переменных привязки;
– динамические *DML* без известного списка переменных привязки;
– динамические *DDL*.
Основные преимущества *NDS* перед DBMS_SQL:
– выполняется быстрее, чем DBMS_SQL;
– синтаксис пакета является зеркальным повторением стандартных статических операторов *SQL*;
– можно вести выборку непосредственно в типы записей *PL/SQL*;
– поддерживает все типы данных *PL/SQL*, в том числе определенные пользователем типы и объекты, а также указатели на наборы строк.

Синтаксический анализ команды *SQL* и ее выполнение за один шаг осуществляет команда EXECUTE IMMEDIATE:

EXECUTE IMMEDIATE *команда_SQL*
[INTO {*переменная* [, *переменная*…] | *запись* | *объектная_переменная*}]
[USING [IN | OUT | IN OUT] *связанный_аргумент*
[, [IN | OUT | IN OUT] *связанный_аргумент* …]]
[{RETURNING | RETURN} INTO *связанный_аргумент* [, *связанный_аргумент*]…];

EXECUTE IMMEDIATE может применяться для любой команды *SQL*, кроме многострочных запросов. В *Oracle*9*i* посредством данной команды можно выполнять массовое извлечение, а также массовую вставку или обновление, даже если при массовой обработке были обнаружены ошибки.

Для применения динамического *SQL* в запросах, возвращающих несколько строк, применяется переменная CURSOR и цикл OPEN FOR.

Команда EXECUTE IMMEDIATE должна завершаться точкой с запятой, тогда как *команда_SQL* не должна содержать завершающей точки с запятой.

Массовые связывания или групповая обработка (FORALL, BULK COLLECT INTO, BULK FETCH, BULK EXECUTE IMMEDIATE, RETURNING INTO, %BULK_ROWCOUNT). Коллекции можно применять для улучшения производительности многократно выполняемых операций *SQL* за счет массового связывания (*bulk binds*). Массовое связывание уменьшает количество переключений контекста от процессора *PL/SQL* к процессору БД и обратно. Кроме того, связывание может применяться для реализации динамического *SQL*. Массовое связывание реализуется двумя конструкциями *PL/SQL*: FORALL и BULK COLLECT INTO.

Рассмотрим синтаксис оператора FORALL:

FORALL *индекс* IN *нижняя_граница..верхняя_граница*
[SAVE EXECEPTONS]
{EXECUTE IMMEDIATE *команды_SQL*};

где *индекс* – целая переменная, может применяться только для *команды_SQL* и только как индекс коллекции; *нижняя_граница..верхняя_граница* – ограничение для элементов коллекции, которые будут обрабатываться процессором БД; SAVE EXEPTIONS – позволяет продолжить массовую обработку даже в случае ошибок в некоторых отдельных операциях.

После завершения обработки цикла подробную информацию о возникших проблемах можно будет получить в атрибуте %BULK_EXEPTIONS.

Рассмотрим поэтапно следующий пример.

1. Создадим таблицу:

```
CREATE TABLE bulk_rowcount_test AS
SELECT * FROM employees;
```

2. Выполним анонимный блок:

```
DECLARE
TYPE t_array_tab IS TABLE OF VARCHAR2(30);
l_array t_array_tab := t_array_tab('Bred', 'Tom', 'Angelina', 'Den');
BEGIN
  -- Выполнить операцию массового удаления
  FORALL i IN l_array.first .. l_array.last
    DELETE FROM bulk_rowcount_test
    WHERE FIRST_NAME = l_array(i);
  -- Сообщить об обработанных строках
  FOR i IN l_array.first .. l_array.last LOOP
    DBMS_OUTPUT.put_line ('Element: ' || RPAD(l_array(i), 15, ' ') ||
      ' Rows affected: ' || SQL%BULK_ROWCOUNT(i));
  END LOOP;
END;
```

Как видно, для операций массового связывания можно использовать атрибут курсора %BULK_ROWCOUNT. Этот атрибут подобен ассоциативному массиву, содержащему количество строк, изменяемых при выполнении команд массового связывания. Элемент %BULK_ROWCOUNT с номером *n* содержит количество строк, измененных *n*-м выполнением команды *SQL*.

Передать %BULK_ROWCOUNT как параметр в другую программу нельзя, как и нельзя использовать агрегатное присваивание другой коллекции. %ROWCOUNT содержит сумму всех элементов %BULK_ROWCOUNT. Атрибуты %FOUND и %NOTFOUND отражают результаты самого последнего выполнения команды *SQL*.

Когда *PL/SQL* выполняет эту команду, на обработку серверу БД отправляется вся коллекция, а не каждый ее элемент по отдельности. Можно применять инструкцию FORALL как BULK COLLECT INTO, обращаясь к синтаксису EXECUTE IMMEDIATE для динамического *SQL*.

BULK COLLECT INTO имеет следующий синтаксис:

команда SQL BULK COLLECT INTO *список имен коллекции*;

где *команда_SQL* – это команда *SQL*, такая как SELECT INTO, FETCH INTO, RETURNING INTO или EXECUTE IMMEDIATE; *список_имен_коллекций* – разделенный запятыми список коллекций, по одной для каждого столбца списка SELECT. Коллекции записей не могут быть предметом назначения инструкций BULK COLECT INTO. Однако сервер *Oracle* поддерживает извлечение набора объектов одного типа и массовую сборку их в коллекцию объектов.

Можно включить инструкцию BULK COLLECT INTO в команды SELECT INTO, FETCH INTO и RETURNING INTO, ее можно применять с динамическим *SQL*, выполняется при помощи команды EXECUTE IMMEDIATE.

Пакетный запрос многих строк. При запросе большого числа строк можно увеличить производительность, если вместо поочередного зачитывания строк результата зачитать их всех сразу, значительно снизив тем самым количество переключений контекста от *PL/SQL* к *SQL* и обратно. Для пакетного чтения необходимо снабдить оператор FETCH инструкцией BULK COLLECT. Данные при этом должны записываться не в переменные, а в ассоциативные коллекции, например:

```
DECLARE
   TYPE GenericCursor IS REF CURSOR;
   c1 GenericCursor;
   TYPE VarcharTable IS TABLE OF VARCHAR2(200) INDEX BY BINARY_INTEGER;
empnames VarcharTable;
BEGIN
   OPEN c1 FOR SELECT lname FROM employees;
   FETCH c1 BULK COLLECT INTO empnames;
   CLOSE c1;
END;
```

Сравним в *SQL*Plus* время работы запроса, возвращающего большой набор данных, с циклами FOR и BULK COLLECT. Для этого сначала создадим таблицу:

```
CREATE TABLE bulk_collect_test AS
SELECT owner, object_name, object_id
FROM all_objects
--WHERE ROWNUM < 2000;
```

При работе в облаке с *iSQL*Plus* нужно убрать комментарий в коде создания таблицы и далее выполнить анонимный блок:

```
DECLARE
  TYPE t_bulk_collect_test_tab IS TABLE OF bulk_collect_test%ROWTYPE;
  l_tab    t_bulk_collect_test_tab := t_bulk_collect_test_tab();
```

```
  l_start  NUMBER;
BEGIN
  -- Время обычного извлечения (regular time)
  l_start := DBMS_UTILITY.GET_TIME;
  FOR cur_rec IN (SELECT * FROM   bulk_collect_test)
  LOOP
    l_tab.EXTEND;
    l_tab(l_tab.last) := cur_rec;
  END LOOP;
  DBMS_OUTPUT.PUT_LINE ('Regular (' || l_tab.count || ' rows): ' ||
                  (DBMS_UTILITY.GET_TIME - l_start));

  -- Врямя пакетного извлечения (bulk time)
  l_start := DBMS_UTILITY.GET_TIME;

  SELECT *
  BULK COLLECT INTO l_tab
  FROM bulk_collect_test;
  DBMS_OUTPUT.PUT_LINE('Bulk    (' || l_tab.count || ' rows): ' ||
 (DBMS_UTILITY.GET_TIME - l_start));
END;
```

В результате могут будут получены, например, данные, представленные ниже, убедительно подтверждающие большую производительность пакетной обработки данных.

```
Regular (7302 rows): 5
Bulk    (7302 rows): 1
```

10.7. Пакет DBMS_OUTPUT

В *PL/SQL* не предусмотрены средства, обеспечивающие ввод/вывод данных в вызывающую среду, так как для работы с данными, хранимыми в базе, выводить значения переменных и структуры данных не нужно.

В состав *PL/SQL* 2.0 был включен пакет DBMS_OUTPUT, обеспечивающий вывод информации, а средств ввода данных в языке *PL/SQL* до сих пор не существует. Однако вводить информацию можно при помощи переменных подстановки *SQL*Plus*. В *PL/SQL* версии 2.3 и выше имеется также пакет UTL_FILE, который используется для чтения информации из файлов операционной системы и для записи информации в файлы.

Владельцем пакета DBMS_OUTPUT является пользователь *Oracle* с именем SYS. В файле-сценарии создания DBMS_OUTPUT роли PUBLIC предоставляется полномочие EXECUTE на этот пакет; кроме того, в данном файле для па-

кета создается общий синоним. Это означает, что любой пользователь *Oracle* может вызывать подпрограммы пакета DBMS_OUTPUT, не указывая перед его именем SYS.

Таким образом, пакет DBMS_OUTPUT предназначен для эмуляции простых действий вывода на экран в языке *PL/SQL* и позволяет выводить на экран строки из *PL/SQL*-кода. Простейший пример вывода:

```
BEGIN
DBMS_OUTPUT.PUT_LINE ('Hello, Oracle!');
END;
```

На самом деле ввод/вывод на экран выполняет утилита *SQL*Plus*, а *PL/SQL*-код выполняется другим процессом, обычно работающим на другой машине в сети. Также отметим, что, по существу, пакет DBMS_OUTPUT реализует алгоритм «первым пришел – первым обслужен».

В пакете реализованы две базовые операции – GET и PUT. Операция PUT берет свои аргументы и помещает во внутренний буфер для хранения. Операция GET считывает этот буфер и возвращает его содержимое процедуре в качестве аргумента.

Все процедуры выдают данные во внутренний буфер, который представляет собой *PL/SQL*-таблицу в теле пакета DBMS_OUTPUT. Выданные результаты буферизуются в этой таблице и не будут видны в среде *SQL*Plus*, пока не завершится выполнение соответствующего *PL/SQL*-кода. Даже после этого результаты можно увидеть, только если используемый клиент «знает» о пакете DBMS_OUTPUT и позаботится о выдаче накопленных результатов. Вывод на экран можно включать и отключать в *SQL*Plus* (в *iSQL*Plus* включение выполняется автоматически) следующим образом:

```
SET SERVEROUTPUT ON

SET SERVEROUTPUT OFF
```

При включении вывода DBMS_OUTPUT происходит не только накопление данных в буфере, но и задается максимальный объем данных, которые можно накопить.

Общая длина выдаваемой строки (сумма всех байт, помещенных в буфер пользователем до вызова процедуры PUT_LINE или NEW_LINE, завершающей эту строку) должна быть не более 255 байт.

По умолчанию создается буфер DBMS_OUTPUT размером 20 000 байт. При достижении этого предела будет выдано сообщение

```
ORA-20000: ORU-10027: buffer overflow, limit of 20000 bytes
```

Если нужно, можно указать другой размер буфера с помощью команды

```
SET SERVEROUTPUT ON SIZE размер_буфера
```

где *размер буфера* задается в диапазоне от 20 000 до 1 000 000 байт и представляет аргумент процедуры DBMS_OUTPUT.ENABLE, вызываемой по умолчанию. Например:

```
SET SERVEROUTPUT ON SIZE 1000000
```

Однако реально можно поместить в буфер меньше данных, чем установленный предел. Пакет DBMS_OUTPUT использует простой алгоритм упаковки данных при помещении в *PL/SQL*-таблицу. Он не помещает *i*-ю строку в *i*-й элемент массива, а плотно упаковывает массив. В первом элементе массива могут оказаться первые пять строк, выданные с помощью этого пакета. Для того чтобы поместить несколько строк в одну, необходимо расходовать дополнительные ресурсы. Эти дополнительные ресурсы для размещения сведений о том, где находятся данные пользователя и какого они размера, выделяются из того же ограниченного пространства. Поэтому, даже выполнив команду SET SERVEROUTPUT ON SIZE 1000000, можно выдать меньше миллиона байт.

При фиксированном размере выдаваемой строки, причем когда все строки одинаковой длины, можно точно рассчитать, сколько байт удастся выдать. Если выдаются строки переменной длины, то вычислить заранее, сколько байт удастся выдать, нельзя.

Отметим, что при использовании пакета DBMS_OUTPUT начальные пробелы в выдаваемых строках удаляются. Ошибочно думать, что это свойство пакета DBMS_OUTPUT, это выполняется утилитой *SQL*Plus*.

Простое решение данной проблемы – использовать расширенный синтаксис команды SET SERVEROUTPUT:

```
SET SERVEROUTPUT  ON|OFF} [SIZE n]
[FORMAT{WRAPPED|WORD_WRAPPED|TRUNCATED}]
```

где WRAPPED означает, что утилита *SQL*Plus* при необходимости переносит на новую строку выданные сервером результаты, начиная с позиции, задаваемой командой SET LINESIZE;

WORD_WRAPPED означает, что переносится каждая строка выданных сервером результатов, начиная с позиции, задаваемой командой SET LINESIZE. Перенос выполняется по словам. Утилита *SQL*Plus* выравнивает каждую строку влево, удаляя все начальные пробелы. Это значение параметра является стандартным;

TRUNCATED означает, что каждая строка результатов сервера усекается до длины, задаваемой командой SET LINESIZE.

Действие каждой опции форматирования проще всего понять последовательно выполняя примеры в *SQL*Plus*:

```
SET LINESIZE 5
SET SERVEROUTPUT ON FORMAT WRAPPED
```

```
BEGIN
DBMS_OUTPUT.PUT_LINE (                    'Hello, Oracle!!!!!');
END;

SET SERVEROUTPUT ON FORMAT WORD_WRAPPED

BEGIN
DBMS_OUTPUT.PUT_LINE (                    'Hello, Oracle!!!!!');
END;

SET SERVEROUTPUT ON FORMAT TRUNCATED

BEGIN
DBMS_OUTPUT.PUT_LINE (                    'Hello, Oracle!!!!!');
END;
```

Выполнение операции PUT обеспечивается процедурами PUT, PUT_LINE и NEW_LINE, а выполнение операции GET – процедурами GET_LINE и GET_LINES. Управляют буфером процедуры ENABLE и DISABLE. Если объявлена процедура DISABLE, то содержимое буфера уничтожается и последующие вызовы PUT и PUT_LINE бесполезны.

Процедуры PUT и PUT_LINE определены в пакете следующим образом:

```
PROCEDURE PUT (a VARCHAR2);
PROCEDURE PUT (a NUMBER);
PROCEDURE PUT (a DATE);

PROCEDURE PUT_LINE (a VARCHAR2);
PROCEDURE PUT_LINE (a NUMBER);
PROCEDURE PUT_LINE (a DATE);
```

Как видно, данные процедуры являются перегруженными по признаку использования различных типов данных параметра.

PUT_LINE добавляет к аргументу символ новой строки, сообщая о конце строки; PUT этого не делает. Вызов PUT_LINE аналогичен вызову PUT с последующим вызовом NEW_LINE.

Процедура GET_LINE определена в пакете следующим образом:

```
PROCEDURE GET_LINE (line OUT VARCHAR2, status OUT INTEGER);
```

где *line* представляет собой последовательность символов, из которых состоит одна строка буфера, а *status* указывает на то, успешно или нет, была считана эта строка. Максимальная длина строки – 255 байт. Если строка считана, то в переменной *status* находится 0, если в буфере больше нет строк для считывания, то в *status* – 1.

Аргументом процедуры GET_LINES является индексная таблица. Тип таблицы и вызов данной процедуры выглядят следующим образом:

```
TYPE CHARARR IS TABLE OF VARCHAR2(255) INDEX BY BINARY INTEGER;
PROCEDURE GET_LINES (lines OUT CHARARR, numlines IN OUT INTEGER);
```

где *numlines* – число запрошенных строк, на входе (параметр IN) в GET_LINES указывается число запрошенных строк, на выходе (параметр OUT) – число фактически возвращаемых строк.

Тип CHARARR определен в пакете DBMS_OUTPUT, поэтому если GET_LINES вызывается явным образом, нужно объявлять переменную с типом DBMS_OUTPUT.CHARARR. Например, предварительно создав таблицу

```
CREATE TABLE temp_table (char_col VARCHAR2(10));
```

выполним следующий анонимный блок:

```
DECLARE
v_Data DBMS_OUTPUT.CHARARR;
v_NumLines NUMBER;
BEGIN

DBMS_OUTPUT.ENABLE (1000000);
DBMS_OUTPUT.PUT_LINE ('Line one');
DBMS_OUTPUT.PUT_LINE ('Line two');
DBMS_OUTPUT.PUT_LINE ('Line three');

v_NumLines:=3;
DBMS_OUTPUT.GET_LINES(v_Data, v_NumLines);
FOR v_Counter IN 1..3
LOOP
INSERT INTO temp_table (char_col) VALUES (v_Data(v_Counter));
END LOOP;
END;
```

10.8. Пакет UTL_FILE

Пакет UTL_FILE обеспечивает ввод/вывод текстовых файлов. С помощью UTL_FILE невозможно осуществлять вывод информации непосредственно в двоичные файлы. Файлы двоичного формата можно считывать при помощи объектов BFILE, которые представляют собой особую форму внешних объектов LOB.

Безопасность. В клиентском *PL/SQL* имеется пакет TEXT_IO, подобный UTL_FILE. Эти пакеты различны в отношении безопасности информации. Файлы, создаваемые с помощью клиентского пакета TEXT_IO, можно размещать в любом месте на клиентском компьютере при наличии необходимых привилегий для работы с операционной системой. При выполнении клиентских операций файлового ввода/вывода каких-либо привилегий для работы с *PL/SQL* и собственно БД не требуется.

На сервере необходимо поддерживать безопасность информации на более высоком уровне. Для этого выделяются специальные каталоги (*accessible directories*), в которые пакет UTL_FILE может записывать данные. Они определяются параметром UTL_FILE_DIR инициализационного файла БД. Каждый доступный каталог указывается в этом файле отдельной строкой:

UTL_FILE_ DIR = *имя_каталога*

Вид спецификации имени каталога зависит от применяемой операционной системы.

Для того чтобы обратиться к файлу с помощью пакета UTL_FILE, нужно передать функции FOPEN имя каталога и имя файла в виде отдельных параметров. Имя каталога сопоставляется со списком доступных каталогов, и, если оно там найдено, операция разрешается (при соблюдении ограничений безопасности операционной системы). Если в FOPEN указано имя недоступного каталога, выдается сообщение об ошибке. Подкаталоги доступных каталогов можно использовать только тогда, когда они также указаны как доступные явным образом.

Операции файлового ввода/вывода выполняются пакетом UTL_FILE в режиме пользователя *Oracle*. Следовательно, пользователь *Oracle* должен иметь привилегии операционной системы на чтение из всех доступных каталогов и на запись в них, а также все файлы, создаваемые в результате работы пакета UTL_FILE, будут принадлежать ему. Кроме того, файлы будут создаваться с привилегиями, установленными операционной системой для пользователя *Oracle* по умолчанию. Если с этими файлами должны работать другие пользователи, обращаясь к ним не из пакета UTL_FILE, системный администратор обязан изменить полномочия на эти файлы.

Если при выполнении некоторой процедуры или функции пакета UTL_FILE возникает ошибка, устанавливается исключительная ситуация. Возможные исключительные ситуации приведены в табл. 10.3. Необходимо обратить внимание на то, что в их состав входят восемь определенных в пакете UTL_FILE и две стандартные исключительные ситуации (NO_DATA_FOUND и VALUE_ERROR). Исключительные ситуации UTL_FILE можно распознать по имени или с помощью обработчика исключительной ситуации OTHERS. Стандартные исключительные ситуации определяются еще и по значениям, возвращаемым для них функцией SQLCODE.

Таблица 10.3

Исключительные ситуации, устанавливаемые в UTL_FILE

Исключительная ситуация	Причина	Источник
INVALID_PATH	Имя каталога или имя файла неверно или недоступно	FOPEN
INVALID_MODE	Для режима файла указана неверная строка символов	FOPEN
INVALID_FILE_HANDLE	Описатель файла не указывает на открытый файл	FCLOSE, GET_LINE, PUT, PUT_LINE, NEW_LINE, PUTF, FFLUSH
INVALID_OPERATION	Файл нельзя открыть так, как это было запрошено. Возможная причина – полномочия операционной системы. Устанавливается также при попытке записи в файл, открытый для чтения, или при попытке чтения файла, открытого для записи	GET_LINE, PUT, PUT_LINE, NEW_LINE, PUTF, FFLUSH
INVALID_MAXLINESIZE	Указанный максимальный размер строки слишком велик или слишком мал	FOPEN
READ_ERROR	Ошибка операционной системы во время операции чтения	GET_LINE
WRITE ERROR	Ошибка операционной системы во время операции записи	PUT, PUT_LINE, PUTF, NEW_LINE, FFLUSH, FCLOSE, FCLOSE_ALL
INTERNAL_ERROR	Неопределенная внутрисистемная ошибка	Все функции
NO_DATA_FOUND	Во время чтения достигнут конец файла	GET_LINE
VALUE_ERROR	Вводимая строка слишком велика для буфера, указанного в GET_LINE	GET_LINE

Открытие и закрытие файлов. Во всех операциях пакета UTL_FILE фигурирует понятие описателя файла. Описатель или логический номер файла (*file handle*) – это значение, которое используется в *PL/SQL* для идентификации файла, подобно идентификатору курсора в пакете DBMS_SQL. Все описатели

файлов имеют тип UTL_FILE.FILE_TYPE, они возвращаются функцией FOPEN и передаются как параметры вида IN в другие подпрограммы UTL_FILE.

Функция FOPEN открывает файл на ввод или на вывод. В любой момент времени файл может быть открыт или только на ввод, или только на вывод. Синтаксис FOPEN:

FUNCTION FOPEN (*location* IN VARCHAR2, *filename* IN VARCHAR2, *open_mode* IN VARCHAR2);
RETURN FILE_TYPE;

Каталог, путь к которому указывается, должен существовать на момент выполнения функции FOPEN. Однако если установить режим OPEN_MODE как 'w', то существующий файл будет перезаписываться. Параметры и возвращаемое значение функции FOPEN описаны в табл.10.4.

Таблица 10.4

Параметры функции FOPEN

Параметр	Тип	Описание
location	VARCHAR2	Путь к каталогу, в котором расположен файл. Если этот каталог не входит в число доступных, то устанавливается UTL_FILE.INVALID_PATH
filename	VARCHAR2	Имя открываемого файла. Если задан режим 'w', то существующий файл перезаписывается
open_mode	VARCHAR2	Используемый режим. Возможные значения: 'r' – чтение текста; 'w' – запись текста; 'a' – добавление текста. Этот параметр не чувствителен к регистру символов. Если установлен режим 'a', а файл не существует, то он создается в режиме 'w'
возвращаемое значение	UTL_FILE.FILE_TYPE	Описатель файла, используемый впоследствии в функциях

В *Oracle*8*i* и выше в пакет UTL_FILE введен дополнительный, переопределенный вариант функции FOPEN:

FUNCTION FOPEN (*location* IN VARCHAR2, *filename* IN VARCHAR2, *open_mode* IN VARCHAR2, *maxlinesize* IN BINARY_INTEGER);
RETURN FILE_TYPE;

Параметры *location, filename* и *open_mode* ведут себя так же, как и в первом варианте FOPEN, а *maxlinesize* используется для указания максимального размера строки файла. Диапазон значений: от 1 до 32 767. Если это значение не указано, максимальный размер строки устанавливается равным 1024. Если *maxlinesize* меньше 1 или больше 32 767, устанавливается исключительная ситуация UTL_FILE.INVALID_MAXLINESIZE.

После окончания чтения из файла или записи в него этот файл должен быть закрыт с помощью процедуры FCLOSE. В результате освобождаются ресурсы, используемые пакетом UTL_FILE для работы с файлом. Описание FCLOSE выглядит следующим образом:

PROCEDURE FCLOSE (*file_handle* IN OUT FILE_TYPE);

Единственным параметром этой процедуры является описатель файла. Все изменения, которые должны быть записаны в файл, завершаются перед его закрытием. В случае ошибки записи устанавливается UTL_FILE.WRITE_ERROR. Если описатель не идентифицирует корректный открытый файл, то устанавливается UTL_FILE.INVALID_FILEHANDLE.

Перед тем как производить операции чтения из файла или записи/перезаписи в файл, необходимо проверить, открыт ли этот файл. Для этих целей в пакете UTL_FILE предусмотрена функция IS_OPEN. Она возвращает TRUE, если указанный файл открыт, и FALSE, если не открыт. Описание функции IS_OPEN:

FUNCTION IS OPEN (*file_handle* IN FILE TYPE)
RETURN BOOLEAN;

Даже в том случае, когда IS_OPEN возвращает TRUE, при работе с данным файлом могут происходить ошибки операционной системы.

Процедура FCLOSE_ALL пакета UTL_FILE закрывает все открытые файлы. Она предназначена для удаления ненужной информации, что особенно полезно в обработчиках исключительных ситуаций:

PROCEDURE FCLOSE_ALL;

Эта процедура не принимает никаких аргументов. Все ожидающие записи изменения сбрасываются на диск перед закрытием файлов. Поэтому в случае ошибки записи FCLOSE_ALL может устанавливать исключительную ситуацию UTL_FILE.WRITE_ERROR.

Файловый вывод. Для вывода данных в файлы используются пять процедур: PUT, PUT_LINE, NEW_LINE, PUTF и FFLUSH. Максимальный размер выходной записи равен 1023 байтам (если не указано иное значение в FOPEN), в том числе 1 байт для символа новой строки.

Описание процедуры PUT в пакете UTL_FILE имеет следующий вид:

PROCEDURE PUT (*file_handle* IN FILE TYPE, *buffer* IN VARCHAR2);

Символ новой строки в файл не вводится. Для того чтобы включить в файл признак конца строки, используется процедура PUT_LINE или NEW_LINE. В случае ошибки записи устанавливается UTL_FILE.WRITE_ERROR. Параметры процедуры PUT описаны в табл. 10.5.

Таблица 10.5

Параметры процедуры PUT

Параметр	Тип	Описание
file_handle	UTL_FILE.FILE_TYPE	Описатель файла, возвращаемый функцией FOPEN. Если описатель некорректен, то устанавливается UTL_FILE.INVALID_FILEHANDLE
buffer	VARCHAR2	Текстовая строка символов, выводимая в файл. Если файл не был открыт в режиме 'w' или 'a', то устанавливается UTL_FILE.INVALID_OPERATION

Процедура NEW_LINE записывает один или несколько признаков конца строки в указанный файл:

```
PROCEDURE NEW_LINE (file_handle IN FILE_TYPE,
                    lines IN NATURAL := 1);
```

Признак конца строки зависит от используемой системы – в разных операционных системах признаки конца строки различны. В случае ошибочной записи устанавливается UTL_FILE.WRITE_ERROR. Параметры процедуры описаны в табл. 10.6.

Таблица 10.6

Параметры процедуры NEW_LINE

Параметр	Тип	Описание
file_handle	UTL_FILE. FILE_TYPE	Описатель файла, возвращаемый функцией FOPEN. Если описатель некорректен, то устанавливается UTL_FILE.INVALID_FILEHANDLE
lines	NATURAL	Число выводимых признаков конца строки. Значение по умолчанию равно 1, что соответствует выводу одного символа новой строки. Если файл не был открыт в режиме 'w' или 'a', то устанавливается UTL_FILE.INVALID_OPERATION

Процедура PUT_LINE выводит указанную последовательность символов в указанный файл, причем данный файл должен быть открыт на запись. После вывода последовательности выводится символ новой строки, определяемый используемой платформой:

```
PROCEDURE PUT_LINE (file handle IN FILE_TYPE,
                    buffer IN VARCHAR2);
```

Процедура PUTF подобна PUT, но позволяет форматировать выходную строку символов. PUTF – это сокращенный вариант функции PRINTF(), применяемой в языке *C* и имеющей аналогичный синтаксис:

```
PROCEDURE PUTF (file_handle IN FILE TYPE,
                format IN VARCHAR2,
                arg1 IN VARCHAR2 DEFAULT NULL,
                arg2 IN VARCHAR2 DEFAULT NULL,
                arg3 IN VARCHAR2 DEFAULT NULL,
                arg4 IN VARCHAR2 DEFAULT NULL,
                arg5 IN VARCHAR2 DEFAULT NULL);
```

Заметим, что аргументы с *arg1* по *arg5* имеют значения по умолчанию, т. е. они необязательны. В строке FORMAT содержится обычный текст, а также могут применяться два специальных символа %s и \n. Каждый экземпляр %s в этой строке замещается одним из необязательных аргументов, а каждый экземпляр \n – символом новой строки. В случае ошибки записи устанавливается UTL_FILE.WRITE_ERROR.

С помощью процедур PUT, PUT_LINE, PUTF и NEW_LINE данные выводятся через буфер. Когда буфер заполняется, данные физически сбрасываются (flush) в файл. Процедура FFLUSH записывает содержимое буфера в файл немедленно, причем выводятся только те строки буфера, которые заканчиваются символом перевода строки, все данные, записанные последними операциями PUT, остаются в буфере. Описание процедуры выглядит следующим образом:

```
PROCEDURE FFLUSH (file_handle IN FILE_TYPE);
```

Файловый ввод. Процедура GET_LINE используется не для записи в файл, а для чтения из файла. Одна строка текста считывается в указанном файле и возвращается параметром BUFFER. Символ новой строки в возвращаемую последовательность не включается. Описание процедуры GET_LINE выглядит следующим образом:

```
PROCEDURE GET_LINE (file_handle IN FILE TYPE,
                    buffer OUT VARCHAR2);
```

Когда из файла считывается последняя строка, устанавливается исключительная ситуация NO_DATA_FOUND. Если строка не помещается в буфер, размер которого указан как фактический параметр, устанавливается VALUE_ERROR. При чтении пустой строки возвращается пустая последовательность символов (NULL). В случае ошибочной записи устанавливается UTL_FILE.READ_ERROR. Максимальный размер входной строки составляет 1022 байта (если параметром *maxlinesize* функции FOPEN не задан другой размер). Параметры процедуры GET_LINE описаны в табл. 10.7.

Таблица 10.7

Параметры функции GET_LINE

Параметр	Тип	Описание
file_handle	UTL_FILE.FILE_TYPE	Описатель файла, возвращаемый функцией FOPEN. Если описатель некорректен, то устанавливается UTL_FILE.INVALID_FILEHANDLE
buffer	VARCHAR2	Буфер, в который записывается строка. Если файл не был открыт на чтение ('r'), то устанавливается UTL_FILE.INVALID_OPERATION

При написании приложений, использующих процедуры и функции пакета UTL_FILE, рекомендуется всегда включать обработчики реальных ошибок, которые могут возникнуть при вызове определенных процедур или функций.

10.9. Пакет DBMS_PIPE

В дополнение к средствам чтения таблиц БД и записи в эти таблицы в *PL/SQL* предлагаются два стандартных пакета, предназначенных для взаимодействия между соединениями, или сеансами (*sessions*): DBMS_PIPE и DBMS_ALERT. Их можно использовать для обмена сообщениями между сеансами, соединенными с одним и тем же экземпляром БД.

Посредством пакета DBMS_PIPE реализуются программные каналы БД (*database pipe*). Канал БД не зависит от операционной системы и будет работать в любой среде, в которой функционирует *Oracle*. Различные сеансы, соединенные с одним и тем же экземпляром *Oracle*, могут посылать и получать сообщения по такому каналу. Канал могут использовать несколько сеансов, получающих сообщения, или получателей (*readers*), и несколько сеансов, посылающих сообщения, или отправителей (*writers*). При этом сеанс может посылать и получать несколько сообщений. Все эти сеансы могут находиться на разных машинах и работать в различных средах выполнения программ *PL/SQL*. Все, что требуется от сеансов, – соединиться с одним и тем же экземпляром *Oracle*, а также иметь возможность выполнять блоки *PL/SQL*.

Программные каналы асинхронны – они функционируют независимо от транзакций. После того как сообщение послано по каналу, вернуть его невозможно, даже если был выполнен откат той транзакции, которая выдала сообщение. Следовательно, их можно использовать для реализации автономных транзакций, которые можно завершать или откатывать независимо от родительской транзакции.

В прикладном программном интерфейсе DBMS_PIPE и для посылки, и для получения сообщений применяется локальный буфер сообщений. Отправитель

упаковывает в буфере группу элементов данных, а затем его содержимое посылается по каналу в буфер сообщений принимающего сеанса. Принимающий сеанс распаковывает буфер и получает в результате реальную информацию. Подробнее процессы посылки и получения сообщений по программному каналу описываются в следующих параграфах.

Посылка сообщений. Как уже указывалось выше, сообщения посылаются в два этапа. Сначала данные упаковываются в локальном буфере сообщений, а затем содержимое буфера посылается по каналу. Упаковка данных осуществляется с помощью процедуры PACK_MESSAGE, а посылка – с помощью функции SEND_MESSAGE.

Процедура PACK_MESSAGE переопределяется различными типами элементов данных. На принимающей стороне канала процедура UNPACK_MESSAGE также переопределяется различными типами. Описание процедуры PACK_MESSAGE выглядит следующим образом:

```
PROCEDURE PACK_MESSAGE (item IN VARCHAR2);

PROCEDURE PACK_MESSAGE (item IN NUMBER);

PROCEDURE PACK_MESSAGE (item IN DATE);
```

Размер буфера равен 4096 байтам. Если общий размер упакованных данных превышает это значение, то генерируется ошибка: *ORA-6558: Buffer in DBMS_PIPE package is full. No more items allowed* (Буфер пакета DBMS_PIPE заполнен. Ввод других элементов запрещен). Для каждого элемента данных в буфере отводится 1 байт для типа данных, 2 байта – для их длины, 1 дополнительный байт для завершения сообщения, а также пространство для самих данных. В *Oracle*8*i* и выше 2 байта необходимы для идентификатора набора символов и 1 дополнительный байт – для формы набора символов.

Когда после одного или нескольких вызовов PACK_MESSAGE локальный буфер сообщений заполняется данными, его содержимое с помощью SEND_MESSAGE посылается по каналу:

```
FUNCTION SEND_MESSAGE
(pipename IN VARCHAR2,
timeout IN INTEGER DEFAULT MAXWAIT,
maxpipesize IN INTEGER DEFAULT 8192)
RETURN INTEGER;
```

Если программный канал еще не создан, то функция SEND_MESSAGE создает его. Каналы можно создавать и с помощью процедуры CREATE_PIPE, которая применяется в *PL/SQL* версии 2.2 и выше. Параметры функции SEND_MESSAGE описаны в табл. 10.8.

Таблица 10.8

Параметры функции SEND_MESSAGE

Параметр	Тип	Описание
pipename	VARCHAR2	Имя канала; имена каналов ограничены 30 символами и не чувствительны к регистру символов
timeout	INTEGER	Задержка в секундах. Если в силу различных причин (указанных возвращаемым кодом) сообщение не может быть послано, то вызов возвращает число секунд после задержки. Значение по умолчанию DBMS_PIPE.MAXWAIT равно 86 400 000 секундам (1000 дней)
maxpipesize	INTEGER	Общий размер канала в байтах. По умолчанию устанавливается равным 8192 байтам (два сообщения максимального размера). Сумма размеров всех сообщений в канале не может превышать этого значения (когда сообщение получается с помощью RECEIVE_MESSAGE, оно удаляется из канала). После создания канала его максимальный размер становится элементом его описания и сохраняется на все время его существования. В разных вызовах SEND_MESSAGE можно указывать разные значения параметра *maxpipesize*. Если новое значение больше существующего размера, то размер канала увеличивается. Если же новое значение меньше, то сохраняется существующий размер

Значения, возвращаемые функцией SEND_MESSAGE, описаны в табл. 10.9.

Таблица 10.9

Значения, возвращаемые функцией SEND_MESSAGE

Возвращаемое значение	Описание
0	Сообщение успешно послано и при вызове RECEIVE_MESSAGE будет считано
1	Вызов задержан. Это может произойти в том случае, если канал слишком заполнен и сообщение передаться не может либо если нельзя установить блокировку данного канала
3	Вызов прерван из-за внутренней ошибки

Получение сообщений. Функции RECEIVE_MESSAGE и NEXT_ITEM_TYPE и процедура UNPACK_MESSAGE пакета DBMS_PIPE используются для приема сообщений, посланных по каналу, а также для распаковки сообщений и превращения их в исходные элементы данных.

Функция RECEIVE_MESSAGE является обратной по отношению к функции SEND_MESSAGE. Она считывает сообщение из канала и помещает его в локальный буфер сообщений. После этого вызывается процедура UNPACK_MESSAGE, которая считывает данные уже из буфера. Описание функции RECEIVE_MESSAGE выглядит следующим образом:

```
FUNCTION RECEIVE_MESSAGE (pipename IN VARCHAR2,
timeout IN INTEGER DEFAULT MAXWAIT)
RETURN INTEGER;
```

Обычно RECEIVE_MESSAGE вызывается из принимающей программы. Если сообщения не ожидаются, то RECEIVE MESSAGE блокируется до тех пор, пока не считает сообщение. Таким образом, сеанс-получатель будет находиться в режиме ожидания до посылки сообщения по каналу. Принимающие программы похожи на процессы-демоны операционной системы. Демон также «засыпает» и «просыпается» только тогда, когда будет получено сообщение по программному каналу. Параметры функции RECEIVE_MESSAGE описаны в табл. 10.10.

Таблица 10.10

Параметры функции RECEIVE_MESSAGE

Параметр	Тип	Описание
pipename	VARCHAR2	Имя канала. Оно должно быть таким же, как и указанное в SEND_MESSAGE. На имя канала налагаются те же ограничения (длина менее 30 символов, нечувствительность к регистру символов)
timeout	INTEGER	Максимальное время ожидания сообщения в секундах. Как и в SEND_MESSAGE, значение по умолчанию – MAXWAIT (1000 дней). Если параметр *timeout* равен 0, то значение RECEIVE_MESSAGE возвращается немедленно с состоянием 0 (сообщение считано) или 1 (задержка)

Коды, возвращаемые функцией RECEIVE_MESSAGE, описаны в табл. 10.11.

Таблица 10.11

Возвращаемые коды функции RECEIVE_MESSAGE

Возвращаемое значение	Описание
0	Сообщение успешно получено. Оно было считано в локальный буфер и может быть распаковано с помощью UNPACK_MESSAGE
1	Задержка. За время ожидания RECEIVE_MESSAGE по каналу не было передано ни одного сообщения
2	Сообщение в канале слишком велико для буфера. Это внутренняя ошибка, которая не происходит в обычных условиях
3	Вызов прерван из-за внутренней ошибки

Функция NEXT_ITEM_TYPE возвращает тип данных следующего элемента в буфере. На основании этого значения можно решить, в какую переменную написать данные. Если тип известен заранее, то вызывать ее необязательно. Функция NEXT_ITEM_TYPE описывается следующим образом:

```
FUNCTION NEXT_ITEM_TYPE RETURN INTEGER;
```

Возвращаемые коды описаны в табл. 10.12.

Таблица 10.12

Возвращаемые коды функции NEXT_ITEM_TYPE

Возвращаемое значение	Описание
0	Элементов в буфере больше нет
6	Следующий элемент в буфере типа NUMBER
9	Следующий элемент в буфере типа VARCHAR2
12	Следующий элемент в буфере типа DATE

Для распаковки принятых сообщений в пакете DBMS_PIPE предназначена процедура UNPACK_MESSAGE. Данная процедура является обратной по отношению к процедуре PACK_MESSAGE. Подобно PACK_MESSAGE, она переопределяется типами считываемых элементов. Описание процедуры UNPACK_MESSAGE выглядит следующим образом:

```
PROCEDURE UNPACK_MESSAGE (item OUT VARCHAR2);
PROCEDURE UNPACK_MESSAGE (item OUT NUMBER);
PROCEDURE UNPACK_MESSAGE (item OUT DATE);
```

В *item* записывается элемент данных из буфера. Если в буфере больше нет данных или если следующий элемент в буфере имеет не тот тип, который был запрошен, генерируется ошибка *Oracle ORA*-6556 или *ORA*-6559. Перед установлением ошибки *PL/SQL* пытается преобразовать тип следующего элемента в нужный тип, используя формат преобразования, заданный по умолчанию.

Создание программных каналов и управление ими. Если канал не существует, то при первом указании его имени в SEND_MESSAGE он создается неявным образом. В *PL/SQL* версии 2.2 и выше каналы можно создавать и удалять явно, с помощью процедур CREATE_PIPE и REMOVE_PIPE.

Программные каналы. Программный канал представляет собой структуру данных, находящуюся в разделяемом пуле системной глобальной области (*SGA*) памяти. Поэтому он не занимает память, в которую могут помещаться другие объекты БД при их считывании с диска. Когда необходимо освободить дополнительное пространство разделяемого пула, каналы автоматически уничтожаются. Однако это касается только тех каналов, в которых не находятся сообщения, ожидающие считывания (хотя канал можно удалить из *SGA* и явным образом, с помощью процедуры PURGE, описанной ниже). Алгоритмом уничтожения программных каналов является алгоритм *LRU*: уничтожается тот канал, который не использовался дольше всех. Максимальный размер канала и, следовательно, размер структуры данных в разделяемом пуле задается параметром *maxpipesize* функций SEND_MESSAGE и CREATE_PIPE.

Общие и частные каналы. Каналы, создаваемые неявно с помощью SEND_MESSAGE, называются общими (*public*). Любой пользователь, имеющий полномочие EXECUTE на пакет DBMS_PIPE и знающий имя канала, может посылать и принимать по нему сообщения. Доступ к частным (*private*) каналам ограничивается пользователем, создавшим этот канал, хранимыми процедурами, выполняющимися на основании набора привилегий владельца этого канала, и пользователями, присоединившимися как SYSDBA или INTERNAL.

В *PL/SQL* версий 2.0 и 2.1 применяются только общие каналы, создаваемые неявно. В *PL/SQL* версии 2.2 и выше можно уже создавать каналы явно, используя для этого функцию CREATE_PIPE. Эта функция является единственным способом создания частного канала, однако при желании ей можно воспользоваться и для создания общего канала. Каналы, создаваемые посредством CREATE_PIPE, остаются в разделяемом пуле до тех пор, пока не будут явно удалены с помощью функции REMOVE_PIPE или пока экземпляр сервера БД не будет остановлен. Такие каналы автоматически не уничтожаются в *SGA*. Описание CREATE_PIPE выглядит следующим образом:

```
FUNCTION CREATE_PIPE (pipename IN VARCHAR2,
                      maxpipesize IN INTEGER DEFAULT 8192,
                      private IN BOOLEAN DEFAULT TRUE)
RETURN INTEGER;
```

Если канал успешно создан, CREATE_PIPE возвращает ноль. Если канал уже существует, а текущий пользователь имеет привилегии, необходимые для доступа к нему, то возвращается ноль и данные, находящиеся в канале, остаются без изменений. Если существует общий канал с указанным именем или частный канал, имеющий то же имя и принадлежащий другому пользователю, то устанавливается ошибка: ORA-23322: *insufficient privilege to access pipe*, т. е. привилегии, недостаточные для доступа к каналу и функция завершается неуспешно. Параметры этой функции описаны в табл. 10.13.

Таблица 10.13

Параметры функции CREATE_PIPE

Параметр	Тип	Описание
pipename	VARCHAR2	Имя создаваемого канала. Имена каналов ограничены 30 символами. Имена, начинающиеся с *ORA$*, зарезервированы для внутреннего использования
maxpipesize	INTEGER	Максимальный размер канала в байтах. Этот же параметр используется в SEND_MESSAGE для неявного создания канала. Значение по умолчанию – 8192 байта. Если SEND_MESSAGE вызывается со значением, превышающим *maxpipesize*, размер канала увеличивается. Если SEND_MESSAGE вызывается со значением меньшим, чем *maxpipesize*, то сохраняется существующий размер
private	BOOLEAN	TRUE для частного канала, в противном случае FALSE. Значение по умолчанию – TRUE. Общие каналы создаются неявно функцией SEND_MESSAGE, поэтому, как правило, не имеет смысла устанавливать для данного параметра значение FALSE

Каналы, создаваемые явно функцией CREATE_PIPE, удаляются с помощью функции REMOVE_PIPE. Если при этом в канале находятся сообщения, то они также удаляются. Эта функция – единственный способ удалить те каналы, которые были созданы явно. Описание функции REMOVE_PIPE выглядит следующим образом:

```
FUNCTION REMOVE_PIPE (pipename IN VARCHAR2)
RETURN INTEGER;
```

Единственным параметром этой функции является имя удаляемого канала. Если канал существует, а текущий пользователь имеет на него привилегии, то канал удаляется и функция возвращает ноль. Если канал не существует, также

возвращается ноль. Если канал существует, но текущий пользователь не имеет привилегий на доступ к нему, то устанавливается ошибка *ORA*-23322 (как и в функции CREATE_PIPE).

С помощью процедуры PURGE можно удалить содержимое канала, не уничтожая его самого. Если канал создан неявно, то, поскольку теперь он пуст, его можно удалить из разделяемого пула. Процедурой PURGE несколько раз вызывается RECEIVE_MESSAGE, поэтому содержимое локального буфера сообщений может быть переписано. PURGE описывается следующим образом:

PROCEDURE PURGE (*pipename* IN VARCHAR2);

Привилегии и безопасность. Для пакета DBMS_PIPE установлены три различных уровня безопасности. Первым уровнем является привилегия EXECUTE на сам пакет. По умолчанию при создании пакета эта привилегия не предоставляется всем пользователям. Поэтому обращаться к пакету DBMS_PIPE смогут только те пользователи, которые имеют системную привилегию EXECUTE ANY PROCEDURE. Для того чтобы разрешить доступ к пакету DBMS_PIPE другим пользователям БД, нужно воспользоваться оператором GRANT и предоставить им привилегии EXECUTE на этот пакет.

Для обеспечения первого уровня безопасности информации привилегию EXECUTE на DBMS_PIPE можно предоставить только определенным пользователям БД. После этого можно создать свой собственный пакет для управления доступом к базовым каналам. Затем привилегию EXECUTE на этот пакет можно предоставить и другим пользователям. Второй уровень безопасности – имя программного канала. Если пользователи не знают этого имени, они не смогут посылать и принимать сообщения по данному каналу. Имя программного канала можно выбрать случайным образом или создать имя, уникальное для двух сеансов, взаимодействующих при помощи канала. Последний способ реализуется с помощью функции UNIQUE_SESSION_NAME и описан ниже.

Самым надежным уровнем безопасности является использование частных каналов. Частный канал доступен только пользователю, создавшему его, и пользователям, присоединившимся как SYSDBA или INTERNAL, поэтому доступ к каналу существенно ограничен. Даже если какой-либо пользователь имеет привилегию EXECUTE на пакет DBMS_PIPE и знает имя программного канала, при попытке доступа он получит сообщение об ошибке *Oracle*: *ORA*-23322: *insufficient privilege to access pipe*, т. е. привилегии, недостаточные для доступа к каналу. Данная ошибка возникает только при создании или удалении канала либо при попытке послать или получить сообщение. Другие вызовы в DBMS_PIPE не обращаются непосредственно к программным каналам.

Лучшим способом использования частных каналов является создание хранимых процедур или функций, которые, в свою очередь, вызывают пакет DBMS_PIPE. Поскольку хранимые подпрограммы выполняются на основании набора привилегий своего владельца, к частным каналам можно обращаться и из хранимых подпрограмм.

Установление протокола связи. Программные каналы используются так же, как и другие низкоуровневые пакеты связи, например *TCP*/*IP*. Пользователь имеет возможность самостоятельно определить способы форматирования данных и их пересылки. Кроме того, пользователь может решить, кто должен получить сообщение. Поэтому сеансы-отправители, дабы воспользоваться гибкостью межсеансовых каналов, должны следовать определенным рекомендациям по организации обмена информацией с сеансами-получателями. Прежде всего между сеансами должны быть строго согласованы правила форматирования сообщений.

Форматирование сообщений. Каждое сообщение, посылаемое по программному каналу, состоит из одного или нескольких элементов данных. Элементы данных заносятся в буфер сообщений с помощью процедуры PACK_MESSAGE, и после этого все содержимое буфера посылается по каналу с помощью функции SEND_MESSAGE. На другой стороне канала буфер принимается функцией RECEIVE_MESSAGE, а элементы данных считываются с помощью функции NEXT_ITEM_TYPE и процедуры UNPACK_MESSAGE.

Операции, выполняемые принимающей программой, обычно зависят от содержимого получаемого сообщения. Так, триггер из следующего примера использует первый пункт данных для форматирования сообщения. По существу, первый элемент данных является кодом операции (*opcode*), или инструкцией, которая сообщает принимающей программе о том, как следует интерпретировать остальные данные.

```
CREATE OR REPLACE TRIGGER LogRSInserts
BEFORE INSERT ON registered_staff
FOR EACH ROW
DECLARE
v_Status INTEGER;

BEGIN
/* Сначала упакуем в буфере описание операции */
DBMS_PIPE.PACK_MESSAGE ('I');

/* Упакуем информацию о текущем пользователе и временную метку*/
DBMS_PIPE.PACK_MESSAGE (USER);
DBMS_PIPE.PACK_MESSAGE (SYSDATE);

/* Упакуем новые значения*/
DBMS_PIPE.PACK_MESSAGE (:NEW.fname);
DBMS_PIPE.PACK_MESSAGE (:NEW.lname);
DBMS_PIPE.PACK_MESSAGE (:NEW.tel_no);
DBMS_PIPE.PACK_MESSAGE (:NEW.address);

/* Пошлем сообщение по каналу 'RSInserts' */
```

```
v_Status:= DBMS_PIPE.SEND_MESSAGE ('RSInserts');

/*Если посылка неудачна, установим ошибку так, чтобы изменение не
 было внесено*/

IF v_Status!=0 THEN
RAISE_APPLICATION_ERROR (-20010, 'LogRSInserts trigger' ||
'couldnot send the message, status =' || v_Status);
END IF;
END LogRSInserts;
```

В зависимости от вида информации в сообщении могут содержаться данные различных типов или разное количество элементов данных. Рекомендуется также помимо других инструкций включать в сообщение инструкцию STOP. Ее можно использовать, к примеру, для сообщения ожидающей программе о том, что нужно отсоединиться от БД и корректно завершить работу. Без такого сообщения ожидающая программа будет уничтожена операционной системой и/или БД, что нежелательно.

Адресация данных. Один и тот же канал могут использовать несколько получателей и отправителей, однако принять сообщение может только один получатель. Более того, какой именно получатель действительно примет сообщение, неизвестно. Поэтому рекомендуется адресовать сообщения конкретной программе-получателю. Это можно сделать, сгенерировав для канала уникальное имя, которое будет использоваться только двумя сеансами – одним получателем и одним отправителем. Для этого применяется функция UNIQUE_SESSION_NAME:

```
FUNCTION UNIQUE_SESSION_NAME RETURN VARCHAR2;
```

Каждый вызов UNIQUE_SESSION_NAME возвращает строку символов, максимальная длина которой – 30 символов. Если эту функцию вызывает один и тот же сеанс БД, то всякий раз будет выдаваться одна и та же строка, уникальная среди всех сеансов, соединенных с БД в конкретный момент времени. Однако если сеанс отсоединяется, то его имя может быть позже задействовано другим сеансом.

Строка символов, возвращаемая UNIQUE_SESSION_NAME, может быть использована в качестве имени канала, что обеспечивает прием сообщения только одним получателем. Для этого существует специальный метод. По каналу с заранее определенным именем посылается начальное сообщение, где указывается имя канала, по которому посылается ответ. Принимающая программа расшифровывает начальное сообщение и посылает ответ по новому каналу, который затем будет использоваться только этими двумя сеансами. С новым каналом будут работать только один получатель и один отправитель, поэтому устраняется неопределенность в вопросе о получателе информации.

Контрольные вопросы и задания к разделу 10

1. Что такое стандартные пакеты, каких типов они могут быть?
2. Какие преимущества дает использование динамического *SQL*?
3. Какой тип привязки используется в СУБД *Oracle*?
4. Какие операторы относятся к встроенному динамическому *SQL*?
5. Назовите в правильном порядке программные компоненты пакета DBMS_SQL, выполняемые при обработке оператора SELECT.
6. Какой стандартный пакет позволяет создавать локальные, частные и публичные каналы?
7. Какая команда записывает сообщение в локальный буфер?
8. Какие параметры имеет команда создания приватного канала?
9. Какой пакет используется для ввода/вывода текстовых файлов?
10. Что необходимо передавать в функцию FOPEN?
11. Какие функции и процедуры пакета DBMS_PIPE участвуют в получении сообщений по каналам?
12. Какими командами можно создавать и удалять частный канал?
13. Для чего используется пакет DBMS_SCHEDULER, из каких частей он состоит?
14. Создайте планировщик для однократного запуска анонимного блока, выполняющего обновление поля *commission_pct* в таблице *employees* на 0,5 %, если оно не пустое.
15. Создайте и выполните динамическую процедуру создания и удаления таблицы, представленную в подразд. 10.3.
16. Создайте и выполните динамическую процедуру обновления адреса, представленную в подразд. 10.4.
17. Создайте и выполните динамическую процедуру поиска сотрудников по заданной должности, представленную в подразд. 10.5.
18. Создайте и выполните пример из подразд. 10.6, где необходимо сравнить время работы запроса, возвращающего большой набор данных, с циклами FOR и BULK COLLECT. По возможности сравните получившейся результат в *SQL*Plus* и в облачной версии *iSQL*Plus*.

11. ХРАНИЛИЩА ДАННЫХ

В заключение рассмотрим современные информационные технологии, которые обеспечивают бизнес-аналитикам и управленцам возможность изучать большие объемы данных при помощи быстрого интерактивного отображения информации на разных уровнях детализации. Основой такой возможности послужила концепция складирования данных и появление хранилищ данных (ХД). БД и ХД являются системами хранения данных, однако они служат разным целям. БД хранит данные, как правило, для определенной бизнес-сферы, помогает выполнять повседневную работу пользователей и подвержена постоянным изменениям. ХД хранит текущие и исторические данные для всего бизнеса относительно стабильно: данные в нем обычно обновляются согласно расписанию, а также могут пополняться за счет внешних источников, например статистических отчетов. ХД используют сервер БД как источник информации, для извлечения данных из БД организаций и имеют дополнительные функции моделирования данных, управления жизненным циклом данных, интеграции источников данных и т. д. Материал подраздела основан на информации в источнике [9].

11.1. Концепция систем складирования данных

Информационная технология складирования данных (*DW – data warehousing*) впервые появилась у компании *IBM* и была окончательно сформулирована Б. Инмоном и Р. Кимбаллом в 1990-х годах как метод решения информационно-аналитических задач в области принятия и поддержки решений. Возникнув на стыке технологии проектирования БД, систем поддержки принятия решений и компьютерного анализа данных, концепция складирования данных оказалась пригодной для широкого круга приложений в бизнесе, науке и технологии.

Во всем мире организации накапливают или уже накопили в процессе своей административно-хозяйственной деятельности большие объемы данных, в том числе и в электронном виде. Эти коллекции данных хранят в себе большие потенциальные возможности по извлечению новой аналитической информации, на основе которой можно и необходимо строить стратегию организации, выявлять тенденции развития рынка, находить новые решения, обусловливающие успешное развитие в условиях конкурентной борьбы.

Системы, построенные на основе информационной технологии складирования данных, обладают рядом характерных особенностей, которые выделяют их как новый класс информационных систем (ИС). К таким особенностям относятся предметная ориентация системы, интегрированность хранимых в ней данных, собираемых из различных источников, инвариантность этих данных во времени, относительно высокая стабильность данных, необходимость поиска компромисса в избыточности данных.

ХД является местом складирования собираемых в системе данных и информационным источником для решения задач анализа данных и принятия решений. Как правило, объем информации в ХД является достаточно большим.

Упрощенно можно сказать, что ХД управляет данными, которые были собраны как из *OLTP*-систем, так и из внешних источников данных, и которые длительный период времени хранятся в системе.

С точки зрения применения концепции в бизнесе, производстве и технологиях следует придерживаться следующего определения:

ХД – это структурно расширяемая вычислительная среда, спроектированная для анализа неизменяемых во времени данных, которые логически и физически преобразованы из различных источников, соответствующая направлениям бизнеса, обновляемая и поддерживаемая длительный период времени, выраженная в простых бизнес-терминах и обобщенная (суммированная) для быстрого анализа.

На практике для реализации ХД используются СУБД, поддерживающие определенную модель данных. Поэтому с точки зрения реализации ХД следует считать БД специальной структуры.

Одной из главных целей создания систем складирования данных является их ориентация на анализ накопленных данных, т. е. структуризация данных в ХД должна быть выполнена таким образом, чтобы данные эффективно использовались в аналитических приложениях.

Главным отличием использования концепции складирования данных является структуризация, систематизация, классификация, фильтрация и т. п. больших массивов электронной информации в виде, удобном для анализа, визуализации результатов анализа и производства корпоративной отчетности.

Одной из проблем при решении задач анализа данных в рамках операционных систем анализа данных, а это в основном системы, основанные на реляционной модели, является низкая производительность обработки запросов, которые готовят данные для последующего анализа. Такие запросы увеличивают нагрузку на процессоры ОС и в целом ухудшают обработку потока транзакций в БД, исходящего от систем операционной обработки данных.

Типы ХД по Б. Инмону различаются на основе классификации по отраслевому принципу применения ХД и бывают следующими:

– финансовые ХД;
– ХД в области страхования;
– ХД для управления персоналом;
– глобальные ХД;
– ХД с возможностями обнаружения новых данных (*Data Mining*);
– ХД в области телекоммуникаций.

Основные элементы концепции складирования данных:

1. Извлечение данных из операционных (транзакционных) систем. Главный элемент концепции складирования данных состоит в том, что к данным, сохраняемым для анализа, может быть обеспечен наиболее эффективный доступ

только при условии выделения их из операционной системы, т. е. данные из операционной системы должны быть вынесены в отдельную систему складирования данных.

2. Необходимость интегрирования данных из нескольких *OLTP*-систем. Системы складирования данных наиболее полезны, когда данные могут быть извлечены более чем из одной *OLTP*-системы и в месте, отличном от места локализации исходных приложений. ХД может очень эффективно воедино собрать данные от конкретных приложений, таких как продажи, маркетинг, финансы, производство, с учетом их накопления, т. е. сохранить временные ряды основных показателей бизнеса – так называемые исторические данные.

3. Система складирования данных не только может работать как эффективная платформа для консолидации данных из различных источников, но и может собирать многократные версии данных из одного приложения. Например, если организация перешла на новое программное обеспечение, то ХД сохранит необходимые данные из предыдущей системы. В этом отношении система складирования данных может служить средством интеграции наследуемых данных, сохраняя преемственность анализа при смене программно-аппаратной платформы *OLTP*-системы.

4. Различия между транзакционной и аналитической обработкой данных. Одной из наиболее важных причин отделения данных для анализа от данных *OLTP*-систем было потенциальное падение производительности обработки запросов при выполнении процессов анализа данных. Высокая производительность и небольшое время ответа – критические параметры *OLTP*-систем. Потерю производительности и объем накладных затрат, связанных с обработкой предопределенных запросов, обычно легко оценить. С другой стороны, запросы для анализа данных в ХД трудно предсказать, и следовательно, для них сложно оценить время выполнения запроса.

5. Данные в системах складирования данных остаются неизменными. Эта характеристика ХД имеет большое значение для отбора типов данных при размещении их в ХД, а также выбор момента времени, когда данные должны быть занесены в ХД. Последнее свойство называется гранулированностью данных.

6. Данные в хранилище данных хранятся значительно более длительное время, чем в *OLTP*-системах. Данные в большинстве *OLTP*-систем архивируются сразу после того, как они становятся неактивными. Например, заказ может стать неактивным после того, как он выполнен; банковский счет может стать неактивным после того, как он был закрыт. Главная причина для архивирования неактивных данных – это производительность *OLTP*-системы (зачем хранить данные, если к ним не обращаются).

7. Логическое преобразование данных *OLTP*-систем и моделирование данных. Проблемы, связанные с логическим преобразованием данных при переносе их в ХД, могут потребовать значительного анализа и усилий проектировщиков. Архитектура системы складирования данных и модели ХД имеют огромное значение для успеха таких проектов. Даже при том, что большинство ХД развернуто

на реляционных БД, некоторые основные принципы реляционных БД сознательно нарушаются при создании логической и физической модели ХД.

Модель данных ХД определяет его логическую и физическую структуру. В отличие от просто архивированных данных в данном случае невозможно обойтись без процедур детального моделирования. Такое моделирование на ранних стадиях проекта системы складирования необходимо для создания эффективной системы, охватывающей данные всех бизнес-процессов и процедур организации.

Процесс моделирования данных должен структурировать данные в ХД в виде, не зависимом от реляционной модели данных системы, которая поставляет эти данные.

8. Модель ХД подстраивается под структуру бизнеса. Логическая модель ХД ориентирована на предметную область, а не на агрегацию логических моделей конкретных приложений. Сущности (объекты), поддерживаемые в ХД, аналогичны сущностям (объектам) бизнеса – таким как клиенты, продукция (товар), заказы и дистрибьютеры. С точки зрения реляционной теории меняется базисный набор функциональных зависимостей, поддерживаемых в БД.

9. Преобразование информации, описывающей состояние объектов в *OLTP*-системе. Перед размещением данных в ХД они должны быть преобразованы (трансформированы). Большинство данных из *OLTP*-системы или иного внешнего источника не могут поддерживаться в ХД. Многие из атрибутов объектов в *OLTP*-системе очень динамичны и постоянно изменяются. Многие из этих атрибутов не загружаются в ХД, другие атрибуты являются статичными во времени и загружаются в ХД. ХД вообще не должно содержать информации об объектах, которые являются динамическими и постоянно находятся в состоянии модификации.

10. Денормализация модели данных. При проектировании ХД нарушаются принципы реляционной теории, а именно разрешается денормализация модели, в частности, для увеличения производительности запросов за счет ограничения межтабличных соединений в запросах. Каждый запрос в реляционных БД имеет свою стоимость выполнения (*cost performance*), которая очень высока в ХД из-за количества обрабатываемых данных в запросе (и межтабличных соединений, число которых растет пропорционально размерности модели).

11. Статичность взаимосвязей в исторических данных. В реляционных БД проще поддерживать динамические взаимоотношения между сущностями бизнеса, в то время как ХД содержит взаимосвязи между сущностями предметной области в заданное время.

12. Физическое преобразование данных приложений источников. Важным моментом в системах складирования данных является физическое преобразование данных. Эти процедуры в складировании данных известны как процессы очистки данных. Процесс очистки данных является наиболее интенсивным и трудоемким в любом проекте создания ХД. В течение процесса физического преобразования данные находятся в некотором промежуточном файле до того, как

будут занесены в ХД. Когда данные собираются из многих приложений, их целостность может быть проверена в течение процесса формирования преобразованных данных до загрузки в ХД.

Термины и имена *атрибутов сущностей*, используемые в *OLTP*-системах, в процессе преобразования данных для ХД преобразуются в универсальные, стандартные термины, принятые для данной сферы бизнеса.

Таким образом, в процессе преобразования данных проектировщик ХД должен привести данные систем-источников к определенным стандартам, а именно:

– стандартизовать наименования атрибутов в ХД;

– определить одинаковые домены для одних и тех же атрибутов различных систем-источников;

– принять соглашения о значениях по умолчанию для пропущенных данных;

– принять соглашения о предопределенных значениях атрибутов.

13. Агрегация и суммирование конкретных данных. При создании системы складирования данных важным моментом является сбор и определение требований пользователей. Как правило, такие требования позволяют оценить число вопросов, на которые система должна давать ответы. Большинство вопросов носят аналитический характер, который подразумевает определенный уровень агрегации и суммирования данных перед получением конечного результата. Во многих современных системах складирования данных при их создании закладывается определенный набор предопределенных и автоматически генерируемых итоговых отчетов и справок.

11.2. Архитектура хранилищ данных

Под архитектурой ХД понимают совокупность программно-аппаратных компонентов, совокупность технологических и организационных решений, предпринимаемых для создания, разработки и функционирования ХД, т. е. выбор аппаратного и программного обеспечения, выбор способов взаимодействия программно-аппаратных компонентов, выбор способа решения проектной задачи по разработке и созданию ХД. Как правило, архитектуру ХД составляют следующие компоненты:

– средства извлечения данных из различных БД *OLTP*-систем, унаследованных систем и других внешних источников данных;

– средства трансформации и очистки данных. Точность существующих данных доставляет немало хлопот организации. Поэтому перед тем, как поместить данные в хранилище, их необходимо привести в порядок, иначе говоря – очистить;

– программное обеспечение БД. Как правило, это высокопроизводительная распределенная СУБД, используемая для структуризации и хранения информации;

– средства для соединения источников данных с хранилищем и клиентов с сервером.

– специальные программные средства проектирования хранилища, средства работы с репозиторием метаданных и собственно средства оперативной аналитики, или *OLAP*-средства.

Основные типы программно-аппаратной архитектуры ХД. На рис. 11.1 приведена типовая концептуальная архитектура ХД. В конкретных решениях по архитектуре ХД некоторые компоненты схемы могут отсутствовать.

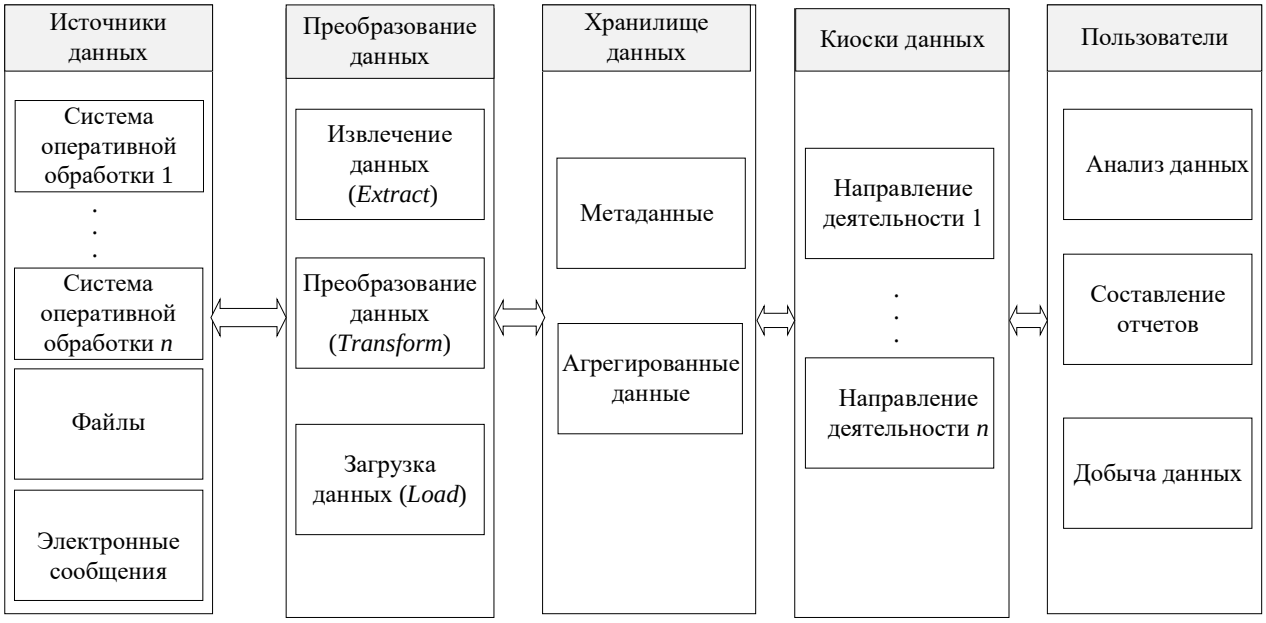


Рис. 11.1. Типовая концептуальная архитектура ХД

Компоненты типовой архитектуры ХД:

1. Программное обеспечение промежуточного слоя. Основное назначение состоит в обеспечении доступа к сети и доступа к данным. Сюда можно отнести сетевые и коммуникационные протоколы, драйверы, системы обмена сообщениями и т. д.

2. БД систем оперативной обработки данных (*OLTP*) и данные внешних источников.

3. Предварительная обработка и загрузка данных. Предварительная обработка, связанная с фильтрацией, очисткой и преобразованием данных из *OLTP*-систем и внешних источников, обычно выполняется в некотором промежуточном файле, который называется иногда загрузочной секцией. После обработки данные загружаются в ХД. Этот компонент включает в себя набор программных средств (*ETL* – средств) для выполнения функций извлечения (*E – extract*), трансформации (*T – transform*), загрузки (*L – load*).

4. Хранилище данных. Представляет собой ядро системы складирования данных. Это могут быть один или несколько серверов БД для поддержки ХД.

5. Метаданные. Представляют собой репозиторий, который играет роль справочника о данных. Он включает терминологию предметной области, сведения об источниках данных, описание источников исходных данных, сведения об алгоритмах обработки исходных данных и т. д.

6. Уровень доступа к данным. Этот компонент включает в себя программное обеспечение, которое обеспечивает взаимодействие конечных пользователей с данным ХД. В настоящее время универсальным средством общения служат *SQL* и его расширения.

7. Уровень информационного доступа. Обеспечивает непосредственное общение пользователя с ХД. В качестве таких средств могут выступать стандартные пакеты *MS Office, Lotus Notes* или специальные программные продукты.

8. Уровень администрирования. Компоненты этого уровня отслеживают выполнение процедур обновления ХД, включающих процедуры подкачки данных, обновления индексов, суммирования и агрегации данных, репликацию данных в распределенной вычислительной среде, авторизацию пользователя и разграничение прав доступа.

Типовыми архитектурами для систем складирования данных принято считать следующие:

– системы с глобальным ХД;

– системы с независимыми киосками данных;

– системы с интегрированными киосками данных;

– системы, разработанные на основе комбинации из вышеперечисленных архитектур.

Глобальное хранилище данных (*Global data warehouse*), или хранилище данных масштаба организации – это такое ХД, в котором будут поддерживаться все данные организации или большая их часть. Это наиболее полное интегрированное ХД с высокой степенью интенсивности доступа к консолидированным данным и использованием его всеми подразделениями организации или руководством организации в рамках основных направлений ее деятельности.

Глобальное ХД может быть физически как централизованным, так и распределенным.

Независимые киоски данных управляются рабочими группами, отделами или направлениями бизнеса и разрабатываются исключительно для реализации аналитических потребностей последних. Вполне возможно, что при этом не существует никакой связи между ними. Например, данные для таких киосков данных могут генерироваться непосредственно в самих подразделениях организации. Данные могут извлекаться из *OLTP*-систем, в частности, при помощи информационных служб организации. Информационные службы могут поддерживать вычислительную среду для киосков данных, но не управляют информацией в них. Данные в киоски могут поступать и из глобального ХД.

Киоски данных могут быть взаимозависимы или взаимосвязаны (так называемые связанные киоски данных). Такая архитектура ХД включает в себя совокупность киосков данных, которые управляются рабочими группами, отделами

или направлениями бизнеса, но разрабатываются в рамках единой для организации схемы удовлетворения информационных и аналитических потребностей. Для взаимосвязанных киосков данных типична распределенная архитектура реализации. Несмотря на то что отдельные киоски данных реализуются в рамках рабочих групп, подразделений и направлений бизнеса, они могут быть интегрированы, т. е. взаимосвязаны, для того чтобы обеспечить представления данных в рамках организации в целом. Фактически на наиболее высоком уровне интеграции они могут стать глобальным ХД. В такой архитектуре пользователи одних подразделений могут получать доступ к данным других подразделений в рамках своих полномочий.

Так же как и для реализации любых типов информационных систем с базами данных, к ХД применимы следующие основные методологические подходы:

– «сверху вниз» (*top down design*);
– «снизу вверх» (*bottom down design*);
– «из середины» (*middle of design*).

Подход «сверху вниз». Подход «сверху вниз» требует детального планирования и проектирования ХД в рамках проекта до начала его выполнения. Это связано с тем, что необходимо привлекать всех потенциальных пользователей ХД для выяснения их информационных потребностей в аналитической обработке данных, принимать решения об источниках данных, безопасности, структурах данных, качестве данных, стандартах данных. Все эти работы должны быть документированы и согласованы, а модель ХД должна быть разработана до начала реализации.

Обычно такой подход практикуют при создании глобального ХД. Если киоски данных включаются в конфигурацию, то они могут быть построены позже.

Достоинством такого подхода является получение более согласованных определений данных и бизнес-правил организации в самом начале работы над созданием ХД. Подход «сверху вниз» хорошо применять в организациях с четко организованной информационно-вычислительной структурой, когда программно-аппаратная платформа определена и существуют слаженно работающие источники данных.

Подход «снизу вверх». При использовании подхода «снизу вверх» начинают с планирования и проектирования киосков данных подразделений без предварительной разработки глобальной информационно-вычислительной инфраструктуры организации. Это не означает, что такая глобальная инфраструктура не будет разработана позже. Такой подход является более приемлемым во многих случаях, поскольку он быстрее приводит к конечным результатам. У него есть и недостатки: данные могут дублироваться и быть несогласованными в разных киосках данных. Чтобы избежать этого, необходимо тщательное планирование и проектирование.

Подход «из середины». Подходы «снизу вверх» и «сверху вниз» могут комбинироваться в зависимости от поставленных перед руководителем проекта по созданию ХД целей. Подход «из середины» представляет собой комбинацию

вышеперечисленных подходов, которые применяются как бы по спирали. Сначала создается ядро системы (подход «сверху вниз»), а затем оно поэтапно наращивается за счет добавления новой или дополнительной функциональности (подход «снизу вверх»). Таким образом, на каждом витке спирали может быть использован каждый из двух указанных выше подходов.

11.3. Характеристика решений ведущих производителей

Решение IBM. Решение компании *IBM* называется *Data Warehouse Plus*. Целью компании в области разработки и поддержки систем складирования данных является обеспечение пользователя интегрированным набором программных продуктов и сервисов в рамках единой архитектуры.

Компания *IBM* предлагает встроенную поддержку трех типов архитектурных решений для ХД:

– независимый киоск данных;

– взаимосвязанные киоски данных;

– глобальное ХД.

Несущая СУБД для ХД – семейство объектно-реляционных СУБД *DB*2. Язык манипулирования данными – *SQL*.

Преимущество решений *IBM* проявляется, когда и системы оперативной обработки данных, и ХД находятся на программном обеспечении *IBM,* т. е. предлагается так называемое замкнутое типовое решение.

С приобретением компании *Informix Software IBM* использует дополнительный ряд удачных решений этой компании в области систем складирования данных.

Решение *Oracle*. Решения, предлагаемые компанией, преследуют две основные цели: предоставление пользователям широкого ассортимента программных продуктов самой компании и деятельность партнеров в рамках программы *Warehouse Technology Initiative*.

Компания *Oracle* не предлагает поддержку каких-либо встроенных архитектурных решений для ХД.

Несущая СУБД для ХД – семейство объектно-реляционных СУБД *Oracle*. Язык манипулирования данными – *SQL*.

Компания выпускает специальный *CASE*-инструментарий для проектирования ХД.

Конкурентные возможности *Oracle* определяются следующими факторами:

– имеется набор готовых приложений для разработки ХД, обеспечивающий полный жизненный цикл;

– компания является одним из лидеров по продажам в области анализа данных;

– совместимость с продуктами, производимыми другими компаниями.

Решение *NCR*. Решение этой компании в области складирования данных ориентировано на организации, у которых имеются потребности в системах *DSS* (система поддержки и принятия решений) и системах *OLAP*. Предлагаемая архитектура называется *Enterprise Information Factory* (виртуальное предприятие).

Несущая СУБД для ХД – реляционная СУБД *Teradata*.

Конкурентным преимуществом решений компании является большой опыт применения СУБД *Teradata* и связанных с ней методов параллельной обработки данных.

Решение *SAS Institute*. Компания считает себя поставщиком полного решения для организации ХД. Она предлагает методологию *Rapid Data Warehousing* для быстрого создания и наполнения ХД. В основу методологии положено:

– обеспечение доступа к данным в ХД с возможностью их извлечения из разнообразных источников данных (интероперабельность);

– преобразование и манипулирование данными в рамках 4*GL* (*Data Step*);

– наличие у компании сервера многомерных БД;

– большой набор программных продуктов компании для аналитической обработки данных и статистического анализа.

Конкурентным преимуществом компании является наличие длинной линейки программных продуктов для статистического и сравнительного анализа данных, который интегрирован в методологию построения и использования ХД.

Решение *Sybase*. Стратегия компании в области ХД основывается на разработанной архитектуре *Warehouse WORKS*.

Несущая СУБД для ХД – реляционная СУБД *Sybase System* и средство подключения к БД *OmniCONNECT*. Язык манипулирования данными – *SQL* и средства быстрой разработки приложений.

Компания выпускает специальный *CASE*-инструментарий для проектирования ХД.

Конкурентным преимуществом компании является наличие набора программных продуктов для обеспечения полного жизненного цикла разработки ХД.

Решение *Microsoft*. Компания сравнительно недавно стала активно предлагать комплексные решения в области ХД. Целью корпорации *Microsoft* является создание инструментальной и технологической среды, которая позволила бы минимизировать затраты на создание ХД и сделала бы этот процесс доступным для массового пользователя. Акцент предлагаемых компанией решений в области складирования данных концентрируется на развитии инструментальных средств *OLAP*.

Корпорация предлагает спецификации среды *Microsoft Data Warehousing Framework* для создания и использования ХД. Открытость среды *Microsoft Data Warehousing Framework* обеспечила ее поддержку многими производителями программного обеспечения.

Цель *Microsoft Data Warehousing Framework* состоит в том, чтобы упростить разработку, внедрение и администрирование решений на основе ХД. Эта спецификация призвана обеспечить:

– открытую архитектуру, которая интегрируется и расширяется третьими фирмами;

– экспорт и импорт гетерогенных данных наряду с их проверкой, очисткой и ведением истории накопления;

– доступ к разделяемым метаданным со стороны процессов разработки ХД.

Несущая СУБД для ХД – реляционная СУБД *MS SQL Server*. Язык манипулирования данными – *SQL* со встроенными средствами обработки многомерных кубов.

Конкурентным преимуществом компании является наличие у нее набора программных продуктов для обеспечения разработки и поддержки ХД, в том числе для очистки данных, при невысокой цене на эти продукты. Ориентация продукции компании на средний и малый бизнес позволяет ей увеличить свои конкурентные преимущества.

Решение *Software AG*. Деятельность компании в области ХД происходит в рамках программы *Open Data Warehouse Initiative*.

Несущая СУБД для ХД – сетевая СУБД *ADABAS*. Язык манипулирования данными – *Natural 4GL*.

У компании имеются собственные средства извлечения и анализа данных, а также программный продукт управления ХД – *SourcePoint*.

Компания имеет сложившийся круг пользователей и долгое время не проявляла инициативы по переходу на распределенные архитектуры, основанные на компьютерах средней мощности.

11.4. Типовые программно-аппаратные решения проектирования хранилищ данных

1. **Виртуальное ХД.** Архитектура обеспечивает доступ к «живым» данным в режиме реального времени через программное обеспечение промежуточного слоя. В основе такого решения лежит репозиторий метаданных, который описывает источники данных, процедуры их предварительной обработки и форматы представления информации конечному пользователю. Недостатки такого решения – интенсивный сетевой трафик, снижение производительности несущей системы, угроза нарушения целостности данных в случае неудачных действий пользователей ХД.

2. **Киоски данных.** Архитектура представляет собой облегченный вариант ХД тематической направленности. Бывают киоски данных, связанные с интегрированным ХД или несвязанные (автономные).

3. **Глобальное ХД.** Архитектура представляет собой единый источник интегрированных данных организации.

4. **ХД с многоуровневой (в основном трехзвенной) архитектурой или корпоративные ХД (*enterprise data warehouse*).** Архитектура является разновидностью глобального ХД, в которой технологически реализуются три уровня, как указано на рис. 11.2. На первом уровне располагается корпоративное ХД организации (*ERP*). На втором уровне поддерживаются связанные киоски данных тематической направленности на основе многомерной СУБД. На третьем уровне находятся клиентские приложения пользователей с установленными на них средствами анализа данных.



Рис. 11.2. ХД с многоуровневой архитектурой

5. **Встроенные (комбинированные) хранилища данных.** Архитектура представляет собой ХД, которые органически встраиваются в виртуальное предприятие (*Enterprise Information Factory, EIF*) или используются как компонент аналитической поддержки в информационной реализации бизнес-функций.

6. **Корпоративная информационная фабрика (*corporate information factory*).** Эта архитектура является развитием архитектуры корпоративного ХД. Ее использование предполагает скоординированное извлечение данных из источников, загрузку их в реляционную БД со структурой в третьей нормальной форме, использование построенного ХД для наполнения дополнительных репозиториев презентационных данных.

7. **ХД с архитектурой шины данных (*data warehouse bus*).** В этой архитектуре ХД не является единым физическим репозиторием. Это «виртуальное» ХД, представляющее коллекцию витрин данных, каждая из которых имеет архитектуру типа «звезда».

8. **Объединенное (федеративное) ХД.** В этой архитектуре ХД состоит из ряда экземпляров ХД, которые функционируют на полуавтономной основе и, как правило, организационно или географически разнесены, однако могут рассматриваться и управляться как одно большое ХД.

Области применения технологии ХД. Концепция ХД находит применение во многих сферах бизнеса, науки и управления. Такие типовые решения использования технологии складирования данных в бизнесе можно разделить на следующие основные группы:

1. Разработка основы для создания аналитических подсистем сопровождения бизнеса.
2. Разработка ХД как составной части виртуального предприятия.
3. Разработка ХД для цифровых (электронных) библиотек и мультимедиа.

Основные сферы применения технологии складирования данных приведены в табл. 11.1.

Таблица 11.1

Области применения концепции складирования данных

Сфера деятельности	Название системы
Сегментация рынка	*CRM*
Планирование продаж, прогнозирование и управление	*CRM, SCM*
Опека клиентов	*CRM*
Схемы лояльности	
Проектирование и разработка продуктов	*MRP/ERP*
Интеграция цепочки поставок	*SCM, ERP/MRP, SCP, SCE, DRP, JIT*
Инновации и новые возможности	
Новые возможности приложений с использованием Интернета/Интранета	*eBusiness, TMP*
Приложения, основанные на агентах программного обеспечения	
Приложения для извлечения знаний и киберорганизация	*EIF*
Распространение *DW* из области стратегического планирования в область бизнес-операций	*VDW*
Приложения для вертикальных секторов индустрии	*CRM, TMP*
Готовые решения	*DW (off-the-shalf)*
Автоматизация принятия решений	*DSS, EIS*
Новые категории оперативных приложений, ориентированные на клиента	*OLAP*
Сбор и анализ экспериментальных данных в химии, физике, биологии	*EDW*
Хранение мультимедийной информации в *DW*	*DL*

Сокращения, использованные в табл. 11.1, имеют следующие значения:

CRM (*Customer Relationship Management*) – управление взаимоотношениями с клиентами;

SCM (*Supply Chain Management*) – управление цепочкой поставок;

SCP (*Supply Chain Planing*) – планирование управления цепочкой поставок;

SCE (*Supply Chain Executing*) – реализация управления цепочкой поставок;

DRP (*Distribution Resource Planing*) – планирование потребностей распределения;

JIT (*Just-in-Time*) – точно в срок;

MRP (*Manufacturing Resource Planing*) – планирование материальных затрат;

VDW (Virtual Data Warehouse) – виртуальные ХД;

DL (*Digital Library*) – цифровые библиотеки;

ERP (*Enterprise Resource Planing*) – системы планирования масштаба предприятия;

TMP (*Trading Partner Management*) – управление деловыми партнерами;

EIF (*Enterprise Information Factory*) – виртуальное предприятие.

Виртуальные предприятия. Одной из перспективных областей применения систем складирования данных является разработка ХД как составной части виртуального предприятия. В этом случае ХД рассматривается как часть интегрированной информационной структуры организации, которая имеет типовую архитектуру, показанную на рис. 11.3.

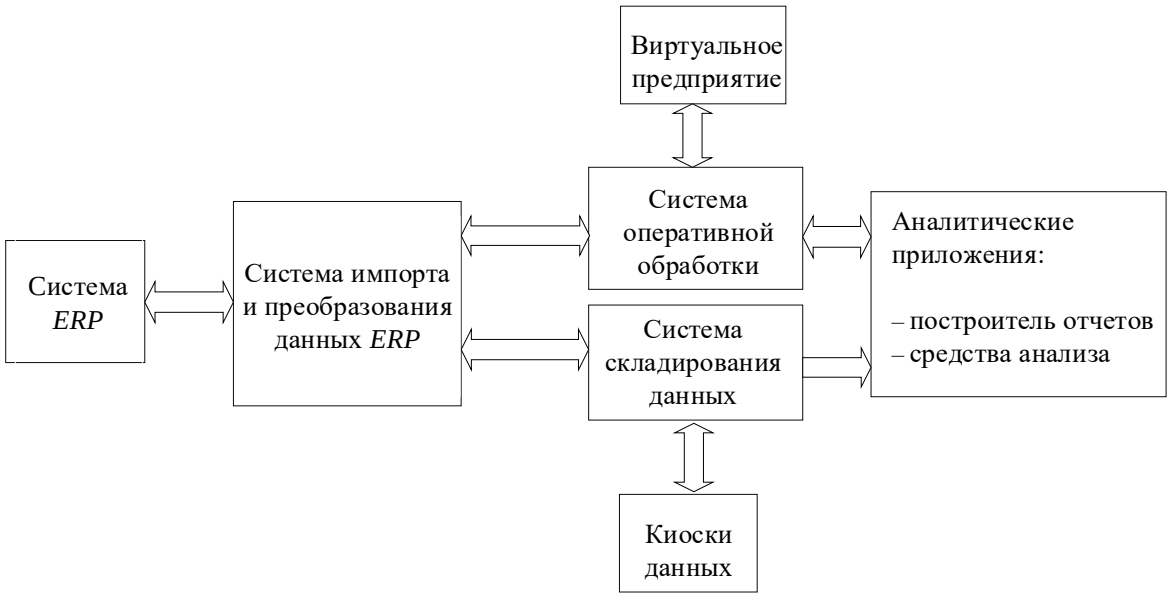


Рис. 11.3. Место ХД в виртуальном предприятии

Мультимедийные ХД. Очень перспективным в последнее время становится разработка ХД для цифровых (электронных) библиотек и мультимедиа. Современные СУБД имеют ряд встроенных возможностей для хранения и выборки мультимедийных данных (например, СУБД *Pilot*). Однако большинство решений по созданию мультимедийных БД реализуется на реляционных СУБД, обладающих возможностью работы с *BLOB*-данными и имеющими поддержку очень больших БД. Типичными представителями таких СУБД являются *Oracle* (имеет специальные средства выборки визуальной информации – *VIR* и интернет-систему обработки файлов – *iFS*), *DB*2 и *Informix* (теперь *IBM*).

Примерами мультимедийных ХД являются разрабатываемые во всем мире электронные хранилища музейных данных (образы картин и других экспонатов).

Корпоративные информационные фабрики. Корпоративная информационная фабрика – это логическая архитектура программно-аппаратного решения по производству, складированию, управлению и доставке данных для поддержки принятия стратегических и тактических решений в масштабе организации. Концепция *CIF*, предложенная классиком в области теории хранилищ данных Б. Инмоном в серии его работ, подразумевала системно организованное взаимодействие репозиториев оперативных данных (*Operational Data Store – ODS*), центрального ХД, витрин данных и системы интеллектуального анализа данных (*Data Mining*) за счет создания технологических цепочек переработки и доставки данных.

На предприятии производственные и финансовые потоки тесно взаимосвязаны с потоками информационными, которые отражают их динамические показатели и текущее состояние. Кроме того, такие информационные потоки являются источником данных для анализа при определении трендов изменений и их количественных характеристик.

Схема превращения данных в информационные продукты, указанная на рис. 11.4, и составляет суть концепции *CIF* на любом предприятии.

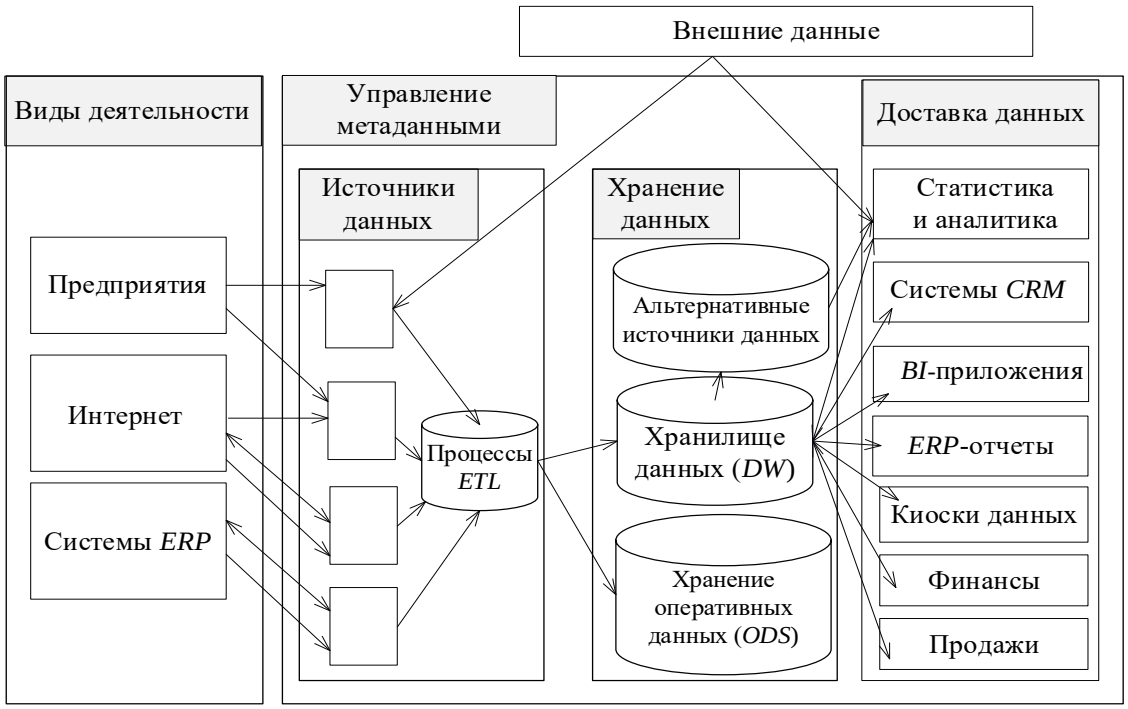


Рис. 11.4. Корпоративная информационная фабрика

На предприятиях основными источниками данных являются *ERP*-системы. Они представляют собой семейство оперативных приложений, обеспечивающих обработку производственных и финансовых данных, включая выполнение бухгалтерских проводок, логистических операций, генерацию текущей оперативной отчетности. Модули *ERP* ориентированы на те информационные продукты, которые они сопровождают или поддерживают, они не предназначены для обработки информации в историческом аспекте и не имеют развитого инструментария для агрегации и систематизации данных предприятия. Из-за строгой предметной направленности у подсистем *ERP*, как правило, слабо развиты взаимосвязи на уровне данных: обычно у них информационный обмен осуществляется небольшими объемами.

В соответствии со словарем Ассоциации операционного менеджмента (*APICS – The Association for Operations Management*) термин *ERP*-система может употребляться в двух значениях:

– *ERP* – это информационная система для идентификации и планирования всех ресурсов предприятия, которые необходимы для осуществления продаж, производства, закупок и учета в процессе выполнения клиентских заказов;

– *ERP* – это методология эффективного планирования и управления всеми ресурсами предприятия, которые необходимы для осуществления продаж, производства, закупок и учета при исполнении заказов клиентов в сферах производства, дистрибуции и оказания услуг.

MRP II – это система планирования производственных ресурсов, включающая в себя функции *MRP* и *CRP* (*Capacity Requirements Planning* – планирование производственных мощностей). Система предназначена для сведения планов продаж и производства, планирования потребностей в сырье, управления производством, планирования производственных мощностей, материально-технического снабжения, планирования ресурсов сбытовой сети, финансового планирования, моделирования и оценки деятельности.

Наиболее распространенные системы класса *ERP*: *SAP R*/3; *Oracle Applications*; *Baan*; Галактика4; Парус; 1С: Предприятие 8; *Dynamics Ax*; *Cognos*; *Navision Attain* и *Navision Axapta*.

Организация процесса интеграции является еще одним фактором успеха в создании *CIF*: информация извлекается из разнородной вычислительной среды *ERP*, преобразуется с целью повышения ее качества и складируется. Все это делается для того, чтобы системы поддержки и принятия решений могли в дальнейшем ее активно использовать.

При наполнении корпоративного ХД в нем обычно предусматриваются инструментальные средства:

– для извлечения и доставки из различных БД и внешних источников;

– очистки, преобразования и интеграции;

– загрузки;

– актуализации.

ХД *CIF* – это предметно-ориентированная, интегрированная, неизменяемая и поддерживающая хронологию коллекция данных, используемая для поддержки принятия решений. С позиций *CIF* хранилище является отправной точкой при преобразовании данных в информационные продукты (аналитические отчеты и пр.). Оно всегда предоставляет своим потребителям проверенные и согласованные данные по всей организации в целом, независимо от источника их происхождения.

Процесс управления данными предусматривает комплекс процедур, отвечающих за прохождение информации в *CIF*. Он включает в себя архивацию и восстановление данных, секционирование, управление перемещением данных в системе, агрегацию и т. д.

В конечном итоге информация должна попасть к потребителю в заданном виде, чтобы послужить базисом для принятия взвешенных управленческих решений. Логично на выходе *CIF* для производства информационных продуктов применять:

– средства для многомерного представления данных и манипулирования ими;

– средства для формирования отчетов;

– систему информационных запросов.

ХД с архитектурой шины данных. В данной архитектуре ХД с архитектурой шины данных, предложенной Р. Кимболлом, первичные данные преобразуются в необходимые структуры на стадии подготовки данных. При этом обязательно принимаются во внимание требования к скорости обработки информации и качеству данных. Подготовка данных начинается со скоординированного извлечения их из источников. Ряд операций совершается централизованно, например поддержание и хранение общих справочных данных, другие действия могут быть распределенными.

В отличие от корпоративной информационной фабрики, в ХД с архитектурой шины данных чаще используются связанные киоски данных, которые разрабатываются для обслуживания бизнес-процессов (бизнес-показателей или бизнес-событий), а не направлений бизнеса. Например, данные о заказах, которые должны быть доступны для общекорпоративного использования, вносятся в ХД только один раз, в отличие от CIF, в котором их пришлось бы трижды копировать в витрины данных отделов маркетинга, продаж и финансов. После того как в ХД появляется информация об основных бизнес-процессах, консолидированные киоски данных могут выдавать их перекрестные характеристики. Матрица шины данных корпоративного ХД с архитектурой шины выявляет и усиливает связи между показателями бизнес-процессов (фактами) и описательными атрибутами (измерениями).

Объединенное (федеративное) ХД. Для любой организации, особенно многофилиальной, наличие согласованной управленческой информации, необходимой для четкого понимания того, как функционирует бизнес, является одной из актуальных задач.

Обычный подход к улучшению информированности о бизнес-операциях – проведение стандартизации «сверху вниз» как структуры отчетности, так и модели данных. Однако с практической точки зрения стандартизация бизнес-структур оказывается для большинства организаций малоэффективной – требуется слишком много средств и времени.

Архитектура федеративного ХД. В этой архитектуре на основе иерархии связанных ХД можно обмениваться данными, бизнес-моделями и структурами отчетности, благодаря чему возможно, с одной стороны, осуществлять общий контроль и предусмотреть определенную степень стандартизации, а с другой – позволить региональным отделениям сохранить автономность и учесть местную специфику.

Федеративное ХД состоит из ряда экземпляров ХД, которые функционируют на полуавтономной основе и, как правило, организационно или географически разнесены, однако могут рассматриваться и управляться как одно большое ХД. Применение такой архитектуры снижает риск неудачи при глобальном развертывании системы, поскольку каждое локальное ХД меньше по масштабу, отвечает местным требованиям бизнеса и может управляться сотрудниками регионального подразделения.

Каждый из экземпляров федеративного ХД хранит копию базовой бизнес-модели и общие основные данные (*common master data*), причем каждое ХД более высокого уровня содержит итоговые транзакционные данные более низкого уровня. Общие основные данные – например, схема организационной структуры компании – отправляются «вниз», т. е. из корпоративного (глобального) ХД, а суммарные данные о транзакциях отправляются «верх», т. е. из локального ХД. Таким образом, «федерация» ХД может предоставить местным отделениям необходимую гибкость, а также обеспечить общий контроль и согласованность; при этом каждое ХД функционирует независимо от всех других.

Для федеративных ХД характерны общая семантика и бизнес-правила, стандартизованный набор процессов извлечения из бизнес-правил, децентрализованные ресурсы и управление, параллельная разработка.

Проект разработки ХД начинается после того, как выбраны инструменты разработки и сформирована команда проекта. В проектный цикл разработки ХД обычно включаются следующие типовые процессы (этапы):

– формулирование требований;
– создание вычислительной среды;
– моделирование данных;
– определение процедур извлечения преобразования и загрузки данных;
– проектирование аналитических отчетов;
– разработка приложений ХД;
– настройка производительности;
– проверка качества;
– передача системы складирования данных в эксплуатацию.

Типовая модель бизнес-процессов разработки ХД представлена в нотации *IDEF*0 на рис. 11.5.

Контрольные вопросы и задания к разделу 11

1. Чем различаются и в чем сходны БД и ХД?
2. Как классифицируются ХД по Б. Инману?
3. Назовите основные элементы концепции складирования данных.
4. Изобразите основные компоненты типовой архитектуры ХД на рисунке.
5. Что из себя представляют и для чего выполняются *ETL*-процессы?
6. Назовите типовые программно-аппаратные решения проектирования ХД.
7. Что обозначает термин *ERP*? Назовите самые известные *ERP*-системы.
8. Каковы области применения концепции складирования данных?
9. Что понимают под термином «киоск данных»?
10. Какие процессы включаются в типовой алгоритм проектирования ХД?



Рис. 11.5. Укрупненная бизнес-модель (*IDEF*0) процесса разработки ХД

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Oracle Database [Электронный ресурс] // Википедия: офиц. сайт. – 2023. – Режим доступа: https://ru.wikipedia.org/?curid=79815&oldid=131002092.

2. Семейство продуктов Oracle [Электронный ресурс] // СУБД Oracle: администрирование, разработка и программирование. – 2016. – Режим доступа: http://oracledb.ru/administrirovanie-subd/obzor-i-arxitektura-subd/semejstvo-produktov-oracle.html.

3. Oracle Apex [Электронный ресурс]. – 2023. – Режим доступа: https://apex.oracle.com.

4. SQL [Электронный ресурс] // Википедия: офиц. сайт. – 2024. – Режим доступа: https://ru.wikipedia.org/?curid=7579&oldid=135891515.

5. Гринвальд, Р. Oracle. Справочник / Р. Гринвальд, Д. К. Крейнс ; пер. с англ. – СПб. : Символ-Плюс, 2005. – 976 с.

6. Фейерштейн, С. Oracle PL/SQL для профессионалов / С. Фейерштейн, Б. Прибыл. – 6-е изд. – СПб. : Питер, 2015. – 1024 с.

7. Код создания схемы HR [Электронный ресурс] // GitHub. – 2014. – Режим доступа: https://github.com/dlee0113/oracle_pl_sql_programming/blob/master/hr_schema_install.sql.

8. Капанов, Н. А. Базы данных САПР. Лабораторный практикум / Н. А. Капанов. – Минск : БГУИР, 2006. – 47 с.

9. Туманов, В. Е. Проектирование хранилищ данных для систем бизнес-аналитики : учеб. пособие / В. Е. Туманов. – М. : Интернет-Ун-т информ. технологий : Бином. Лаборатория знаний, 2010. – 615 с.

10. Прайс, Дж. Oracle Database 11g. Операторы SQL и программы PL/SQL / Дж. Прайс ; пер. с англ.– СПб. : Лори, 2014. – 690 с.

11. Скотт, У. Oracle Database 10g. Программирование на языке PL/SQL / У. Скотт, Р. Хардман, М. МакЛафлин. – М. : Лори, 2007. – 816 с.

12. Кайт, Т. Oracle для профессионалов. Архитектура, методики программирования и особенности версий 9i, 10g и 11g / Т. Кайт ; пер. с англ. – М. : Лори, 2011. – 842 с.

13. Коннолли, Т. Базы данных: проектирование, реализация и сопровождение. Теория и практика / Т. Коннолли, К. Бегг. – 3 изд., перераб. – М. : Вильямс, 2003. – 1440 с.

14. Крупская, М. А. Информационное обеспечение систем управления. Лабораторный практикум : учеб.-метод. пособие / М. А. Крупская. – Минск : БГУИР, 2019. – 147 с.

Св. план 2024, поз. 1

Учебное издание

Крупская Марина Александровна

ОСНОВЫ ПРОГРАММИРОВАНИЯ С ИСПОЛЬЗОВАНИЕМ *SQL* И *PL/SQL*

УЧЕБНОЕ ПОСОБИЕ

Редактор *Ю. В. Граховская*
Корректор *Е. Н. Батурчик*
Компьютерная правка, оригинал-макет *Е. Г. Бабичева*

Подписано в печать 04.06.2025. Формат 60×84 1/16. Бумага офсетная. Гарнитура «Таймс».
Отпечатано на ризографе. Усл. печ. л. 17,55. Уч.-изд. л. 18,8. Тираж 100 экз. Заказ 2.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий №1/238 от 24.03.2014,
№2/113 от 07.04.2014, №3/615 от 07.04.2014.
Ул. П. Бровки, 6, 220013, г. Минск