

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Методические указания

по выполнению лабораторных работ
по дисциплине «РАСПРЕДЕЛЕННАЯ ОБРАБОТКА ДАННЫХ»
для студентов направления подготовки
09.03.03 Прикладная информатика

(ЭЛЕКТРОННЫЙ ДОКУМЕНТ)

Ставрополь 2026

Оглавление

Лабораторная работа1. Общее введение, типы данных и средства определения доменов.....	4
Введение.....	4
Типы данных SQL	9
Средства определения, изменения определения и отмены определения доменов	19
Неявные и явные преобразования типа или домена	23
Заключение	26
Лабораторная работа2. Средства определения базовых таблиц и ограничений целостности.....	26
Введение	26
Средства определения, изменения и ликвидации базовых таблиц	28
Средства определения и отмены общих ограничений целостности.....	42
Заключение	48
Лабораторная работа3. Общая характеристика оператора SELECT и организация списка ссылок на таблицы в разделе FROM	48
Введение.....	48
Скалярные выражения	49
Общая структура оператора выборки в языке SQL	56
Заключение	65
Лабораторная работа4. Предикаты раздела WHERE оператора SELECT	65
Введение.....	65
Логические выражения раздела WHERE	66
Заключение	90
Лабораторная работа5. Группировка и условия раздела HAVING, порождаемые и соединенные таблицы	91
Введение.....	91
Агрегатные функции, группировка и условия раздела HAVING	94
Ссылки на порождаемые таблицы в разделе FROM.....	105
Более сложные конструкции оператора выборки	108
Заключение	122
Лабораторная работа6. Средства формулировки аналитических и рекурсивных запросов	123
Введение.....	123
Возможности формулирования аналитических запросов	125
Рекурсивные запросы.....	133
Заключение	140
Лабораторная работа7. Средства манипулирования данными	140

Введение.....	140
Базовые средства манипулирования данными	141
Представления, над которыми возможны операции обновления.....	147
Операции обновления баз данных и механизм триггеров.....	161
Заключение	171
Лабораторная работа8. Средства языка SQL для обеспечения авторизации доступа к данным, управления транзакциями, сессиями и подключениями	171
Введение.....	171
Поддержка авторизации доступа к данным в языке SQL.....	172
Управление транзакциями в SQL	184
Подключения и сессии.....	193
Заключение	197
Лабораторная работа9. Объектные расширения	200
Введение.....	200
Определяемые пользователями типы	206
Типизированные таблицы	213
Заключение	223

Лабораторная работа 1. Общее введение, типы данных и средства определения доменов

Введение

В начале лекции мы представим небольшой исторический обзор SQL. Язык уже далеко не молод. В 2004 г. сообщество баз данных отметило его 30-летний юбилей. Поэтому, чтобы правильно понимать и трактовать современные варианты SQL, нужно знать историю языка хотя бы в общих чертах.

Краткая история языка SQL

Язык SQL, предназначенный для взаимодействия с базами данных, появился в середине 70-х гг. (первые публикации датируются 1974 г.) и был разработан в компании IBM в рамках проекта экспериментальной реляционной СУБД *System R*. Исходное название языка *SEQUEL* (*Structured English Query Language*) только частично отражало суть этого языка. Конечно, язык был ориентирован главным образом на удобную и понятную пользователям формулировку запросов к реляционным БД. Но, в действительности, он почти с самого начала являлся полным языком БД, обеспечивающим помимо средств формулирования запросов и манипулирования БД следующие возможности:

- средства определения и манипулирования схемой БД;
- средства определения *ограничений целостности* и *триггеров*;
- средства определения представлений БД;
- средства определения структур физического уровня, поддерживающих эффективное выполнение запросов;
- средства *авторизации доступа* к отношениям и их полям ¹;
- средства определения точек сохранения транзакции, и выполнения фиксации и *откатов* транзакций.

В языке отсутствовали средства явной синхронизации доступа к объектам БД со стороны параллельно выполняемых транзакций: с самого начала предполагалось, что необходимую синхронизацию неявно выполняет СУБД.

В настоящее время язык SQL реализован во всех коммерческих реляционных СУБД и почти во всех СУБД, которые изначально основывались не на реляционном подходе. Все компании-производители провозглашают соответствие своей реализации стандарту SQL, и на самом деле реализованные диалекты SQL очень близки. Этого удалось добиться не сразу.

Наиболее близки к *System R* были две системы компании IBM – *SQL/DS* и *DB2* ². Разработчики обеих систем использовали опыт проекта *System R*, а СУБД *SQL/DS* напрямую основывалась на программном коде *System R*. Отсюда предельная близость диалектов SQL, реализованных в этих системах, к *SQL System R*. Из *SQL System R* были удалены только те части, которые были недостаточно проработаны (например, точки сохранения) или реализация которых вызывала слишком большие технические трудности (например, ограничения целостности и триггеры). Можно назвать этот путь к коммерческой реализации SQL движением сверху вниз.

Другой подход применялся в таких системах, как *Oracle*, *Informix* и *Sybase*. Несмотря на различие в способах разработки систем, реализация SQL везде происходила "снизу вверх". В первых выпущенных на рынок версиях этих систем использовалось ограниченное

подмножество *SQL System R*. В частности, в первой известной нам реализации *SQL* в СУБД *Oracle* в операторах выборки не допускалось использование вложенных *подзапросов* и отсутствовала возможность формулировки запросов с соединениями нескольких отношений.

Тем не менее, несмотря на эти ограничения и на очень слабую, на первых порах, эффективность СУБД, ориентация компаний на поддержку разных аппаратных платформ и заинтересованность пользователей в переходе к реляционным системам позволили компаниям добиться коммерческого успеха и приступить к совершенствованию своих реализаций. В текущих версиях *Oracle*, *Informix*, *Sybase* и *Microsoft SQL Server* поддерживаются достаточно мощные диалекты *SQL*, хотя реализация иногда вызывает сомнения.³

Особенностью большинства современных коммерческих СУБД, затрудняющей сравнение существующих диалектов *SQL*, является отсутствие единообразного описания языка. Обычно описание разбросано по разным руководствам и перемешано с описанием специфических для данной системы языковых средств, не имеющих прямого отношения к *SQL*. Тем не менее можно сказать, что базовый набор операторов *SQL*, включающий операторы определения схемы БД, выборки и манипулирования данными, *авторизации доступа к данным*, поддержки встраивания *SQL* в языки программирования и операторы *динамического SQL*, в коммерческих реализациях устоялся и более или менее соответствует стандарту.

Деятельность по стандартизации языка *SQL* началась практически одновременно с появлением его первых коммерческих реализаций. В 1982 г. комитету по базам данных Американского национального института стандартов (*ANSI*) было поручено разработать спецификацию стандартного языка *реляционных баз данных*. Первый документ из числа имеющихся у автора проектов стандарта датирован октябрём 1985 г. и является уже не первым проектом стандарта *ANSI*. Стандарт был принят *ANSI* в 1986 г., а в 1987 г. одобрен Международной организацией по стандартизации (*ISO*). Этот стандарт принято называть ***SQL/86***.

Понятно, что в качестве основы стандарта нельзя было использовать *SQL System R*. Во-первых, этот вариант языка не был должным образом технически проработан. Во-вторых, его слишком сложно было бы реализовать (кто знает, как бы сложилась судьба *SQL*, если бы все идеи проекта *System R* были реализованы полностью). Поэтому за основу был взят диалект языка *SQL*, сложившийся в *IBM* к началу 1980-х гг. В сущности, этот диалект представлял собой технически проработанное подмножество *SQL System R*.

К 1989 г. стандарт *SQL/86* был несколько расширен, и был подготовлен и принят следующий стандарт, получивший название ***ANSI/ISO SQL/89***. Анализ доступных документов показывает, что процесс стандартизации *SQL* происходил очень сложно с использованием не только научных доводов. В результате *SQL/89* во многих частях имеет чрезвычайно общий характер и допускает очень широкое толкование. В этом стандарте полностью отсутствуют такие важные разделы, как манипулирование схемой БД и *динамический SQL*. Многие важные аспекты языка в соответствии со стандартом определяются в реализации.⁴

Возможно, наиболее важными достижениями стандарта *SQL/89* являются четкая стандартизация синтаксиса и семантики операторов *выборки данных* и манипулирования данными и фиксация средств ограничения целостности БД. Были специфицированы средства определения первичного и внешних *ключей отношений* и так называемых

проверочных *ограничений целостности*, которые представляют собой подмножество немедленно проверяемых *ограничений целостности SQL System R*. Средства определения *внешних ключей* позволяют легко формулировать требования так называемой *ссылочной целостности* БД. Это распространенное в реляционных БД требование можно было сформулировать и на основе общего механизма *ограничений целостности SQL System R*, но формулировка на основе понятия внешнего ключа более проста и понятна.

Осознавая неполноту стандарта *SQL*, на фоне завершения разработки этого стандарта специалисты различных компаний начали работу над стандартом *SQL2*. Эта работа также длилась несколько лет, было выпущено множество проектов стандарта, пока наконец в марте 1992 г. не был принят окончательный проект стандарта (*SQL/92*). Этот стандарт существенно полнее стандарта *SQL/89* и охватывает практически все аспекты, необходимые для реализации приложений: манипулирование схемой БД, управление транзакциями (появились точки сохранения) и сессиями (сессия – это последовательность транзакций, в пределах которой сохраняются временные отношения), подключения к БД, *динамический SQL*. Наконец, были стандартизованы отношения-каталоги БД, что вообще-то не связано непосредственно с языком, но очень сильно влияет на реализацию.⁵

В 1995 г. стандарт был дополнен спецификацией интерфейса уровня вызова (Call-Level Interface – ***SQL/CLI***). *SQL/CLI* представляет собой набор спецификаций интерфейсов процедур, вызовы которых позволяют выполнять динамически задаваемые операторы *SQL*. По сути дела, *SQL/CLI* представляет собой альтернативу *динамическому SQL*. Интерфейсы процедур определены для всех основных языков программирования: *C, Ada, Pascal*, *PL/1* и т. д. Следует заметить, что стандарт *SQL/CLI* послужил основой для создания повсеместно распространенных сегодня интерфейсов *ODBC (Open Database Connectivity)* и *JDBC (Java Database Connectivity)*.

В 1996 г. к стандарту *SQL/92* был добавлен еще один компонент – ***SQL/PSM (Persistent Stored Modules)***. Основная цель этой спецификации состоит в том, чтобы стандартизировать способы определения и использования *хранимых процедур*, т. е. специальным образом оформленных программ, включающих операторы *SQL*, которые сохраняются в базе данных, могут вызываться приложениями и выполняются внутри СУБД.

Незадолго до завершения работ по определению стандарта *SQL2* была начата разработка стандарта *SQL3*. Первоначально планировалось завершить проект в 1995 г. и включить в язык некоторые объектные возможности: *определяемые пользователями типы данных*, поддержку *триггеров*, поддержку темпоральных свойств данных и т. д. Реально работу над новым стандартом удалось частично завершить только в 1999 г., и по этой причине (а также в связи с проблемой 2000 года) стандарт получил название *SQL:1999*.

Приведем краткую характеристику текущего состояния *стандарта SQL:1999* и перспектив его развития. Прежде всего, заметим, что каждый новый вариант *стандарта языка SQL* был существенно объемнее предыдущих версий. Так, если *стандарт SQL/89* занимал около 600 страниц, то объем *SQL/92* составлял на 300 с лишним страниц больше. Самые первые проекты *SQL3* занимали около 1500 страниц. Это вполне естественно, потому что язык усложняется, а его спецификации становятся более детальными и точными. Но разработчики *SQL3* пришли к выводу, что при таких объемах стандарта вероятность его принятия и последующей успешной поддержки заметно уменьшается. Поэтому было принято решение разбить стандарт на относительно

независимые части, которые можно было бы разрабатывать и поддерживать по отдельности.

В 1999 г. были приняты пять первых частей *стандарта SQL:1999*. Первая часть (SQL/Framework) посвящена описанию концептуальной структуры стандарта. В этой части приводится развернутая аннотация следующих четырех частей и формулируются требования к реализациям, претендующим на соответствие стандарту.

Вторая часть SQL:1999 (SQL/Foundation) образует *базис* стандарта. Вводится *система типов языка*, формулируются правила определения *функциональных зависимостей* и *возможных ключей*, определяются синтаксис и семантика основных операторов SQL:

- операторов определения и манипулирования *схемой базы данных*;
- операторов манипулирования данными;
- операторов *управления транзакциями*;
- *операторов управления подключениями к базе данных* и т. д.

Третью часть занимает уточненная по сравнению с SQL/92 спецификация SQL/CLI. В четвертой части специфицируется SQL/PSM – синтаксис и семантика языка определения *хранимых процедур*. Наконец, в пятой части – SQL/Bindings – определяются правила связывания SQL для стандартных версий языков программирования FORTRAN, COBOL, PL/1, Pascal, Ada, C и MUMPS.

В *стандарт SQL:1999* должны были войти еще несколько частей. Среди них спецификации следующих средств:

- управление *распределенными транзакциями* (SQL/Transaction);
- поддержка темпоральных свойств данных (SQL/Temporal);
- управление внешними данными (SQL/MED);
- связывание с объектно-ориентированными языками программирования (SQL/OLB);
- поддержка оперативной аналитической обработки (SQL/OLAP).

В конце 2003 г. был принят и опубликован новый вариант международного **стандарта SQL:2003**. Многие специалисты считали, что в варианте стандарта, следующем за SQL:1999, будут всего лишь исправлены неточности SQL:1999. Но на самом деле, в SQL:2003 специфицирован ряд новых и важных свойств, часть из которых мы затронем в этом курсе.

Претерпела некоторые изменения общая организация стандарта. *Стандарт SQL:2003* состоит из следующих частей:

- 9075-1, SQL/Framework;
- 9075-2, SQL/Foundation;
- 9075-3, SQL/CLI ;
- 9075-4, SQL/PSM ;
- 9075-9, SQL/MED ;
- 9075-10, SQL/OLB ;
- 9075-11, SQL/Schemata ;
- 9075-13, SQL/JRT ;

- 9075-14, *SQL/XML*.

Части 1-4 и 9-10 с необходимыми изменениями остались такими же, как и в *SQL:1999* (разд. 7.4). Часть 5 (*SQL/Bindings*) перестала существовать; соответствующие спецификации включены в часть 2. Раздел части *2SQL:1999*, посвященный информационной схеме, выделен в отдельную часть 11. Появились две новые части – 13 и 14. Часть 13 полностью называется "*SQL Routines and Types Using the Java Programming Language*" ("Использование подпрограмм и типов SQL в языке программирования Java"). Появление такой части стандарта оправдано повышенным вниманием к языку Java со стороны ведущих производителей SQL-ориентированных СУБД. Наконец, последняя часть *SQL:2003* посвящена спецификациям языковых средств, позволяющих работать с *XML-документами* в среде SQL.

На мой взгляд, текущее *состояние процесса* стандартизации языка SQL отражает текущее состояние технологии SQL-ориентированных баз данных. Ведущие поставщики соответствующих СУБД (сегодня это компании *IBM, Oracle* и *Microsoft*) стараются максимально быстро реагировать на потребности и конъюнктуру рынка и расширяют свои продукты все новыми и новыми возможностями. Очевидна потребность в стандартизации соответствующих языковых средств, но процесс стандартизации явно не поспевает за происходящими изменениями.

Структура языка SQL

В данной лекции мы начинаем *систематически* описывать базовые механизмы языка SQL. Чтобы пояснить, о какой части языка пойдет речь в этой и следующих лекциях, обратимся к [рис. 11.1](#).

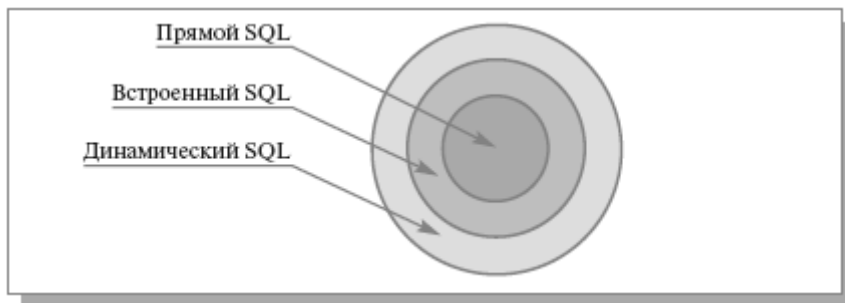


Рис. 11.1. Один из способов разделения языка SQL на уровни

Язык SQL, соответствующий последним стандартам *SQL:2003, SQL:1999* (и даже *SQL/92*), это очень богатый и сложный язык, все возможности которого трудно сразу осознать и тем более понять. Поэтому приходится разбивать язык на уровни, или слои, такие, что каждый уровень языка включает все конструкции, входящие в более низкие уровни. В стандарте определяется несколько способов разбиения языка на уровни. В одной из классификаций язык разбивается на "*базовый*" (*entry*), "*промежуточный*" (*intermediate*) и "*полный*" (*full*) уровни.

Эта классификация ориентирована, прежде всего, на производителей СУБД, в которых поддерживается SQL. Реализация *базового уровня языка* является обязательным условием хотя бы какого-то соответствия стандарту. Реализация *промежуточного уровня* желательна, и обычно именно такой уровень языка поддерживается ведущими компаниями-производителями SQL-ориентированных СУБД. Наконец, *полный уровень*

языка является целью, к достижению которой следует стремиться. В данной классификации критерием отнесения той или иной возможности языка к некоторому уровню является оцениваемая создателями стандарта SQL (большая часть которых является сотрудниками ведущих компаний, производящих SQL-ориентированные СУБД) техническая сложность реализации этой возможности. Конечно, такая классификация важна и для программистов приложений баз данных, но только для того, чтобы оценить реальные возможности конкретной СУБД. Для понимания языка SQL это разбиение на уровни несущественно.

Другая классификация показана на [рис. 11.1](#). Среди всех конструкций языка SQL можно выделить такие конструкции, которые можно было использовать при "прямом" (*direct*) взаимодействии конечного пользователя с СУБД (например, в интерактивном режиме). В некотором смысле этот уровень также является базовым, поскольку соответствующие средства языка в наибольшей степени отражают его ориентированность на работу с множествами. На следующем уровне, уровне "встраиваемого" (*embedded*) SQL, язык расширяется конструкциями, позволяющими использовать возможности прямого SQL в программах, написанных на традиционных языках программирования. Наконец, на уровне "динамического" (*dynamic*) SQL во встраиваемый SQL добавляются конструкции, позволяющие приложениям обращаться к СУБД с конструкциями прямого SQL, которые динамически образуются во время выполнения программы.

Нам кажется, что вторая классификация является более полезной для читателя, постигающего основы языка SQL. По нашему мнению, дополнительные возможности, присутствующие во встраиваемом и динамическом SQL, не слишком сильно влияют на модельное представление языка. Конечно, возможности встраиваемого и динамического SQL необходимо хорошо знать разработчикам приложений SQL-ориентированных баз данных. Но поскольку задачей этого курса не является обучение использованию языка SQL при программировании приложений баз данных, мы не будем затрагивать эти темы. Обратимся к прямому SQL, причем не в полном объеме стандартов SQL:2003 и SQL:1999 (этого не позволяет сделать объем курса). Обсудим только наиболее важные аспекты.

В этой лекции обсуждаются основные аспекты системы типов данных языка SQL и средства определения доменов.

Замечание: Лекции, посвященные языку SQL, опираются, главным образом, на стандарт SQL:1999. В тех случаях, когда будут упоминаться дополнительные возможности, специфицированные в наиболее свежей версии стандарта – SQL:2003, мы будем явно на это указывать. Поэтому здесь мы используем терминологию стандарта (таблицы, строки, столбцы и т. д.).⁶

Типы данных SQL

Данные, хранящиеся в столбцах таблиц SQL-ориентированной базы данных, являются типизированными, т. е. представляют собой значения одного из типов данных, предопределенных в языке SQL или определяемых пользователями путем применения соответствующих средств языка. Для этого при определении таблицы каждому ее столбцу назначается некоторый тип данных (или домен), и в дальнейшем СУБД должна следить, чтобы в каждом столбце каждой строки каждой таблицы присутствовали только допустимые значения. В этом разделе мы обсудим систему типов языка SQL.

Все допустимые в SQL типы данных, которые можно использовать при определении столбцов⁷, разбиваются на следующие категории⁸:

- *точные числовые типы* (exact numerics);
- *приближенные числовые типы* (approximate numerics);
- *типы символьных строк* (character strings);
- *типы битовых строк* (bit strings)⁹;
- *типы даты и времени* (datetimes);
- *типы временных интервалов* (intervals);
- *булевский тип* (Booleans);
- *типы коллекций* (collection types);
- *анонимные строчные типы* (anonymous row types);
- *типы, определяемые пользователем* (user-defined types);
- *ссылочные типы* (reference types).

В столбцах таблиц, определенных на любых типах данных, наряду со значениями этих типов, допускается сохранение *неопределенного значения*, которое обозначается ключевым словом **NULL**. В языке определено, что результатом выражений вида **x a_op NULL**, **NULL a_op x**, **NULL a_op NULL** является **NULL** для всех арифметических операций **a_op** (+, - и т. д.), допустимых для типа данных выражения **x** (выражение **NULL a_op NULL** является допустимым для любой арифметической операции **a_op**). Также по определению полагается, что значением выражений **x comp_op NULL**, **NULL comp_op x**, **NULL comp_op NULL** для всех операций сравнения (=, $\neq$, >, < и т. д.), определенных для типа выражения **x**, является третье логическое значение **unknown**¹⁰ (выражение **NULL comp_op NULL** является допустимым для любой операции сравнения **comp_op**).

Точные числовые типы

К категории *точных числовых типов* в SQL относятся те типы, значения которых точно представляют числа. Типы данных этой категории распадаются на две части: *истинно целые типы* (**INTEGER** и **SMALLINT**) и *типы, допускающие наличие дробной части* (**NUMERIC** и **DECIMAL**). Охарактеризуем эти типы данных более подробно.

Истинно целые типы

- Тип **INTEGER**. Тип служит для представления целых чисел. Точность чисел (число сохраняемых бит) определяется в реализации. При *определении столбца* данного типа достаточно указать просто **INTEGER**.
- Тип **SMALLINT**. Тип также служит для представления целых чисел. Точность определяется в реализации, но она не должна быть больше точности типа **INTEGER**. При *определении столбца* указывается просто **SMALLINT**.¹¹
- *Литералы типов* целых чисел представляются в виде строк символов, изображающих десятичные числа; в начале строки могут присутствовать символы " + " или " - " (если символ знака отсутствует, подразумевается " + "). Примеры *литералов типов* **INTEGER** и **SMALLINT**: 1826545, 876.

Точные типы, допускающие наличие дробной части

- Тип **NUMERIC**. На самом деле, это не просто тип данных, а параметризуемый тип. При *определении столбца* можно указать спецификацию **NUMERIC (p, s)**, где **p** и **s** – *литералы истинно целого типа*, и **p** задает точность значений (число сохраняемых бит), а **s** – шкалу (число десятичных цифр в дробной части). Задаваемая шкала не должна быть

отрицательной и не должна превышать значение точности. При *определении столбца* можно использовать сокращенные формы *спецификации типа* – **NUMERIC** и **NUMERIC (p)**. Первая форма предполагает использование точности, определяемое по умолчанию в реализации, и шкалы, равной нулю, а вторая – использование заданной точности и шкалы, равной нулю. Допустимые диапазоны значений **p** и **s** определяются в реализации.

- Тип **DECIMAL**. Этот тип аналогичен типу **NUMERIC**. Отличие состоит в том, что если при *определении столбца* типа **DECIMAL** задается точность **p**, то на самом деле используется точность **m**, определяемая в реализации, такая, что $m > p$. Шкала всегда устанавливается такой, как явно или неявно (по умолчанию) задается. При указании типа столбца можно использовать спецификации **DECIMAL**, **DECIMAL (p)** и **DECIMAL (p, s)**.
- *Литералы типов точных чисел, допускающих наличие дробной части*, представляются в виде строк символов, изображающих десятичные числа, в начале которых могут присутствовать символы "+" или "-" (если символ знака отсутствует, подразумевается "+"), а внутри последовательности цифр может присутствовать символ ".". Примеры *литералов типов NUMERIC и DECIMAL*: 125, 26.36.

Приближенные числовые типы

К категории *приближенных числовых типов* в SQL относятся те типы, значения которых представляют числа приближенным образом. Приближенные числа представляются в виде пары <мантисса, порядок>, где мантисса состоит из значащих цифр числа, а порядок определяет реальный размер числа. В реализациях *приближенным числовым типам* SQL обычно соответствуют типы с плавающей точкой. В SQL поддерживаются три варианта *приближенных числовых типов*.

- Тип **REAL**. Значения типа соответствуют числам с плавающей точкой одинарной точности. Точность определяется в реализации, но обычно совпадает с точностью одинарной плавающей арифметики, поддерживаемой на аппаратной платформе, которая используется реализацией. При *определении столбца* указывается просто **REAL**.
- Тип **DOUBLE PRECISION**. Точность значений этого типа определяется в реализации, но она должна быть больше точности типа **REAL**. Обычно приближенным числам SQL с двойной точностью соответствуют поддерживаемые аппаратурой числа с плавающей точкой двойной точности. При *определении столбца* указывается просто **DOUBLE PRECISION**.
- Тип **FLOAT**. Это параметризуемый тип, значение параметра **p** которого задает необходимую точность значений. Требуется, чтобы реально обеспечиваемая реализацией точность значений была не меньше **p**. Допустимый диапазон значений параметра **p** определяется в реализации. При *определении столбца* можно указать либо **FLOAT (p)**, либо просто **FLOAT**. В последнем случае подразумевается точность, определяемая реализацией по умолчанию.
- *Литералы приближенных числовых типов* представляются в виде *литерала точного числового типа*, за которым могут следовать символ "E" и *литерал целого числового типа*. Примеры *литералов приближенных числовых типов*: 123, 123.12, 123E12, 123.12E12. Литеральное выражение **xEy** представляет значение $x \cdot (10^y)$.

Типы символьных строк

В SQL определены три параметризуемых типа символьных строк: **CHARACTER** (или **CHAR**), **CHARACTER VARYING** (или **CHAR VARYING**, или **VARCHAR**) и **CHARACTER LARGE OBJECT** (или **CLOB**).¹²

- Тип **CHARACTER**. Значениями типа являются *символьные строки*. Конкретный набор допустимых символов определяется в реализации, но, как правило, включает набор символов ASCII. При *определении столбца* допускается использование спецификаций **CHARACTER (x)** и просто **CHARACTER**. Последний вариант эквивалентен заданию **CHARACTER (1)**. После определения столбца типа **CHARACTER (x)** СУБД будет резервировать место для хранения **x** символов этого столбца во всех строках соответствующей таблицы. Если, например, определен столбец типа **CHARACTER (8)** и в некоторой строке таблицы в него заносится *символьная строка* длиной пять символов, то реально будут храниться восемь символов, последние три из которых будут пробелами¹³.
- Тип **CHARACTER VARYING**. При *определении столбца* допускается использование спецификаций **CHARACTER VARYING (x)** и просто **CHARACTER VARYING**. Последний вариант эквивалентен заданию **CHARACTER VARYING (1)**. Если в некоторой таблице определяется столбец типа **CHARACTER VARYING (x)**, то в каждой строке этой таблицы значения данного столбца будут занимать ровно столько места, сколько требуется для сохранения соответствующей *символьной строки* (но ни одна такая строка не может состоять более чем из **x** символов).¹⁴
- Определен ряд операций, которые можно выполнять над *символьными строками*. Перечислим некоторые из них.
 - Операция *конкатенации* (обозначается в виде " || ") возвращает *символьную строку*, произведенную путем *соединения строк-операндов* в том порядке, в каком они заданы.
 - Функция *выделения подстроки* (**SUBSTRING**) принимает три аргумента – строку, номер начальной позиции и длину – и возвращает строку, выделенную из строки-аргумента в соответствии со значениями двух последних параметров.
 - Функция **UPPER** возвращает строку, в которой все строчные буквы строки-аргумента заменяются прописными. Функция **LOWER**, наоборот, заменяет в заданной строке все прописные буквы строчными.
 - Функция *определения длины* (**CHARACTER_LENGTH**, **OCTET_LENGTH**, **BIT_LENGTH**) возвращает длину заданной *символьной строки* в символах, октетах или битах (в зависимости от вида вычисляющей функции) в виде целого числа.
 - Функция *определения позиции* (**POSITION**) определяет первую позицию в строке **S**, с которой в нее входит заданная строка **S1** (если не входит, то возвращается значение нуль).
- Тип **CHARACTER LARGE OBJECT**. Этот тип данных предназначен для *определения столбцов*, хранящих большие и разные по размеру группы символов. При *определении столбца* задается спецификация **CLOB (z)**, где **z** задает максимальный размер соответствующей группы символов. Максимально возможное значение параметра **z** определяется в реализации, но, очевидно, что оно должно быть существенно больше максимально возможного значения параметра **x**, присутствующего в типах **CHAR** и **CHAR VARYING**.¹⁵
- *Литералы типов символьных строк* представляются в виде последовательностей символов, заключенных в одинарные или двойные кавычки. В первом случае среди набора символов *литерала* допускается наличие символов двойной кавычки, а во втором – символов одинарной кавычки. Примеры *литералов символьных строк*: 'ABCDEF', 'Ab"Ctd', "Fbcdef", "ab'cdtF".

Типы битовых строк

В SQL определены три параметризуемых типа битовых строк: **BIT**, **BIT VARYING** и **BINARY LARGE OBJECT** (или **BLOB**).

- Тип **BIT**. Значениями типа являются *битовые строки*. При *определении столбца* допускается использование спецификаций **BIT (x)** и просто **BIT**. Последний вариант эквивалентен заданию **BIT (1)**. После определения столбца типа **BIT (x)** СУБД будет резервировать место для хранения *x* бит этого столбца во всех строках соответствующей таблицы.
- Тип **BIT VARYING**. При *определении столбца* допускается использование только спецификации без умолчания вида **BIT VARYING (x)**, где значение *x* определяет максимальную длину *битовой строки*, которую можно хранить в данном столбце.
- Над *битовыми строками* определен ряд операций. Некоторые из них мы рассмотрим.
 - Битовая конкатенация (обозначается в виде **||**), которая возвращает результирующую *битовую строку*, полученную путем конкатенации строк-аргументов в том порядке, в котором они заданы.
 - Функция извлечения подстроки из *битовой строки*. Синтаксис и семантика этой функции идентичны синтаксису и семантике функции **SUBSTRING** для *символьных строк*, за исключением того, что первый аргумент и возвращаемое значение являются *битовыми строками*.
 - Функция определения длины (**OCTET_LENGTH**, **BIT_LENGTH**) возвращает длину заданной *битовой строки* в *октетах* или битах в зависимости от выбранной функции.
 - Функция определения позиции (**POSITION**) определяет первую позицию в *битовой строке S*, с которой в нее входит строка *S1*. Если строка *S1* не входит в строку *S*, возвращается значение нуль.
- Тип **BINARY LARGE OBJECT**. Этот тип данных предназначен для *определения столбцов*, хранящих большие и разные по размеру группы байтов. При *определении столбца* задается спецификация **BLOB (z)**, где *z* задает максимальный размер соответствующей группы байтов. С технической точки зрения типы **CLOB** и **BLOB** очень похожи. Их разделение требуется для того, чтобы подчеркнуть, что значения типа **CLOB** состоят из символов (в частности, в них может осмысленно производиться текстовый поиск), а значения типа **BLOB** состоят из произвольных байтов, не обязательно кодирующих символы.
- *Литералы типов битовых строк* представляются как заключенные в одинарные кавычки последовательности символов "0" и "1", предваряемые символом "B"; или предваряемые символом "X" последовательности символов, которые изображают шестнадцатеричные цифры (за цифрой "9" следуют "A", "B", "C", "D", "E" и "F").
Примеры *литералов типов битовых строк*: **B'011100111100011111111'**, **X'78FBCD0012FFFA'**.¹⁶

Типы даты и времени

Возможность сохранения в базе данных информации о дате и времени очень важна с практической точки зрения. Достаточно вспомнить взбудоражившую весь мир "проблему 2000 года", одним из основных источников которой было некорректное хранение дат в базах данных. В стандарте SQL поддержке средств работы с датой и временем уделяется большое внимание. В частности, поддерживаются специальные "темпоральные" типы данных **DATE**, **TIME**, **TIMESTAMP**, **TIME WITH TIME ZONE** и **TIMESTAMP WITH TIME ZONE**. Коротко обсудим эти типы.

Тип даты

- Тип **DATE**. Значения этого типа состоят из компонентов-значений года, месяца и дня некоторой даты. Значение года состоит из четырех десятичных цифр и соответствует летоисчислению от Рождества Христова до 9999 г. Значение месяца состоит из двух десятичных цифр и варьируется от 01 до 12. Значение номера дня месяца состоит из двух десятичных цифр и варьируется от 01 до 31, хотя значение месяца даты может

накладывать ограничения на возможность использования значений дня месяца 29, 30 и 31. В стандарте SQL не накладываются какие-либо ограничения на внутренний способ представления дат, используемый в реализации. При *определении столбца* типа **DATE** указывается просто **DATE**.

- *Литералы* типа **DATE** представляются в виде строки " 'yyyy-mm-dd' ", где символы **y**, **m** и **d** должны изображать десятичные числа. Например, *литерал* **DATE** '1949-04-08' представляет дату 8 апреля 1949 г.

Типы времени

- Тип **TIME**. Значения этого параметризованного типа состоят из компонентов-значений часа, минуты и секунды некоторого времени суток. Значение часа состоит ровно из двух десятичных цифр и варьируется от 00 до 23. Значение минуты состоит из двух десятичных цифр и варьируется от 00 до 59. Основное значение секунды также состоит из двух цифр, но может включать дополнительные цифры, представляющие доли секунды. Так что в целом значение секунды варьируется от 00 до 61.999... В значении времени присутствуют две лишние секунды, поскольку Всемирная служба времени иногда добавляет две секунды к последней минуте года для синхронизации мирового времени с реальным. Решение о поддержке этих "високосных" секунд принимается на уровне реализации. Число цифр в доле секунды также определяется в реализации. В стандарте требуется только то, чтобы это число было не меньше шести. При *определении столбца* типа **TIME** может указываться **TIME (p)** (значение **p** задает точность долей секунды) или просто **TIME** (в этом случае доли секунды не учитываются).
- *Литералы* типа **TIME** представляются в виде строки **TIME 'hh:mm:ss:f...f'**, где символы **h**, **m**, **s** и **f** должны изображать десятичные числа. Например, *литерал* **TIME** '16:33-20:333' представляет время суток 16 часов 33 минуты 20 и 333 тысячных секунды.

Типы временной метки

- Тип **TIMESTAMP**. Значения этого параметризованного типа состоят из компонентов — значений года, месяца и дня некоторой даты, а также компонентов — значений часа, минуты и секунды некоторого времени суток (т. е. каждое значение задает некоторую абсолютную временную метку — отсюда название типа **TIMESTAMP**). Число десятичных цифр в значениях-компонентах и ограничения этих значений такие же, как у значений типов **DATE** и **TIME**. При *определении столбца* типа **TIMESTAMP** может указываться **TIMESTAMP (p)** (значение **p** задает точность долей секунды) или просто **TIMESTAMP** (в этом случае, в отличие от типа данных **TIME**, по умолчанию принимается, что в доли секунды используются шесть десятичных цифр). Максимально допустимое значение **p** определяется в реализации.
- *Литералы* типа **TIMESTAMP** представляются в виде строки **TIMESTAMP 'yyyy-mm-dd hh:mm:ss:f...f'**, где символы **y**, **m**, **d**, **h**, **m**, **s** и **f** должны изображать десятичные числа. Например, *литерал* **TIMESTAMP** '1949-04-08 16:33-20:333' представляет временную метку 16 часов 33 минуты 20 и 333 тысячных секунды 8 апреля 1949 г.

Типы времени и временной метки с временной зоной

- Тип **TIME WITH TIME ZONE**. Этот тип данных похож на тип **TIME** с тем лишь отличием, что значения типа **TIME WITH TIME ZONE** включают дополнительный компонент — значение, характеризующее смещение соответствующего времени относительно гринвичского времени (теперь его называют **UTC** — universal time coordinated). Деталей представления этого дополнительного компонента мы касаться не будем.
- Тип **TIMESTAMP WITH TIME ZONE**. Этот тип данных отличается от типа **TIMESTAMP** тем, что значения типа **TIMESTAMP WITH TIME ZONE** включают дополнительный компонент-значение, характеризующее смещение соответствующего времени относительно гринвичского.

Типы временных интервалов

Вообще говоря, *временным интервалом* называется разность между двумя значениями даты или времени. В SQL определены две категории *типов временных интервалов*: "год-месяц" и "день-время суток". *Временные интервалы* языка SQL не привязываются к начальному и/или конечному значению даты/времени, а описывают только протяженность во времени. В общем случае при *определении столбца типа временного интервала* указывается *INTERVAL start (p) [TO end (q)]*, где в качестве "start" и "end" могут задаваться *YEAR, MONTH, DAY, HOUR, MINUTE* и *SECOND*. Параметр *p* задает требуемую точность лидирующего поля интервала (число десятичных цифр). Параметр *q* может задаваться только в том случае, когда в качестве *end* используется *SECOND*, и указывает точность долей секунды. Если говорить более точно, возможны следующие вариации *типов временных интервалов*.

- Типы категории "год-месяц". Можно определить столбцы следующих типов: *INTERVAL YEAR, INTERVAL YEAR (p)* (значения этих типов – *временные интервалы* в годах), *INTERVAL MONTH, INTERVAL MONTH(p)* (значения этих типов – *временные интервалы* в месяцах), *INTERVAL YEAR TO MONTH, INTERVAL YEAR (p) TO MONTH* (значения этих типов – *временные интервалы* в годах и месяцах). Если значение параметра *p* не указывается явно, по умолчанию принимается его значение "2".
- Типы категории "день-время суток". При *определении столбца* можно использовать следующие комбинации (для полноты перечислим все возможности):
 - *INTERVAL DAY (p),*
 - *INTERVAL DAY,*
 - *INTERVAL DAY (p) TO HOUR,*
 - *INTERVAL DAY TO HOUR,*
 - *INTERVAL DAY (p) TO MINUTE,*
 - *INTERVAL DAY TO MINUTE,*
 - *INTERVAL DAY (p) TO SECOND (q),*
 - *INTERVAL DAY TO SECOND (q),*
 - *INTERVAL DAY (p) TO SECOND,*
 - *INTERVAL DAY TO SECOND,*
 - *INTERVAL HOUR (p),*
 - *INTERVAL HOUR, INTERVAL HOUR (p) TO MINUTE,*
 - *INTERVAL HOUR TO MINUTE,*
 - *INTERVAL HOUR (p) TO SECOND (q),*
 - *INTERVAL HOUR TO SECOND (q),*
 - *INTERVAL HOUR TO SECOND,*
 - *INTERVAL MINUTE (p),*
 - *INTERVAL MINUTE,*
 - *INTERVAL MINUTE (p) TO SECOND (q),*
 - *INTERVAL MINUTE TO SECOND (q),*
 - *INTERVAL MINUTE (p) TO SECOND,*
 - *INTERVAL MINUTE TO SECOND,*
 - *INTERVAL SECOND (p, q),*
 - *INTERVAL SECOND (p),*
 - *INTERVAL SECOND.*

Если значение параметра *p* не указывается явно, по умолчанию принимается его значение "2". Значением параметра *q* по умолчанию является "6".

- Приведем только один пример *литерала* одной из разновидностей типа ***INTERVAL: INTERVAL '10:20' MINUTE TO SECOND*** – *временной интервал* в 10 минут и 20 секунд.
- Над значениями темпоральных типов могут выполняться арифметические операции, смысл которых определяется следующей таблицей:

Тип первого операнда	Операция	Тип второго операнда	Тип результата
<i>Datetime</i>	-	<i>Datetime</i>	<i>Interval</i>
<i>Datetime</i>	+ или -	<i>Interval</i>	<i>Datetime</i>
<i>Interval</i>	+	<i>Datetime</i>	<i>Datetime</i>
<i>Interval</i>	+ или -	<i>Interval</i>	<i>Interval</i>
<i>Interval</i>	* или /	<i>Numeric</i>	<i>Interval</i>
<i>Numeric</i>	*	<i>Interval</i>	<i>Interval</i>

Значения *типов данных временных интервалов* образуются при вычитании одного значения *типа даты* или времени суток из другого значения соответствующего типа. При добавлении интервального значения к значению *типа даты/времени* образуется новое значение *типа даты/времени*. Кроме того, значение интервального типа можно умножать и делить на числовые значения, получая новое значение интервального типа.

Булевский тип

При определении *столбца булевского типа* указывается просто спецификация ***BOOLEAN***. **Булевский тип** состоит из трех значений: true, false и unknown (соответствующие *литералы* обозначаются ***TRUE, FALSE и UNKNOWN***)¹⁷ Поддерживается возможность построения *булевских выражений*, которые вычисляются в *трехзначной логике*. *Таблицы истинности основных логических операций* показаны на [рис. 11.2](#).

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

NOT	
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

Рис. 11.2. Таблицы истинности основных логических операций в трехзначной логике
Типы коллекций

Начиная с *SQL:1999*, в языке поддерживается возможность использования типов данных, значения которых являются коллекциями значений некоторых других типов. Обычно под термином *коллекция* понимается одно из следующих образований: массив, список, множество и *мультимножество*. В варианте *SQL:1999*, принятом в 1999 г., были специфицированы только *типы массивов*. В новом стандарте *SQL:2003* появилась спецификация *типа мультимножества*.

Типы массивов

Любой возможный *тип массива* получается путем применения *конструктора типов* **ARRAY**. При *определении столбца*, значения которого должны принадлежать некоторому *типу массива*, используется конструкция **dt ARRAY [mc]**, где **dt** специфицирует некоторый допустимый в *SQL* тип данных, а **mc** является *литералом* некоторого *точного числового типа* с нулевой длиной шкалы и определяет максимальное число элементов в значении *типа массива* (в терминологии *SQL:1999* это значение называется максимальной кардинальностью массива). В стандарте *SQL:1999* многомерные массивы и массивы массивов не поддерживались. Однако в стандарте *SQL:2003* это ограничение было снято, и теперь типом элементов любого *типа коллекций* может быть любой допустимый в *SQL* тип данных, кроме самого конструируемого *типа коллекции*.

Элементам каждого значения *типа массива* соответствуют их порядковые номера, называемые индексами. Значение индекса всегда должно принадлежать отрезку **[1, mc]**. Значениями *типа массива* **dt ARRAY [mc]** являются все массивы, состоящие из элементов типа **dt**, максимальное значение индекса которых **cs** не превосходит значения **mc**. При сохранении в базе данных значения *типа массива* занимает столько памяти, сколько требуется для сохранения **cs** элементов. Обеспечивается доступ к элементам массива по их индексам. В частности, можно объявить столбец типа **INTEGER ARRAY [10]** и при вставке строки в соответствующую таблицу задать значение только пятого элемента массива. Тогда в строку будет занесен массив из пяти элементов, причем первые четыре элемента будут содержать *неопределенное значение* (**NULL**).

Основными операциями над массивами являются выборка значения элемента массива по его индексу, изменение некоторого элемента массива или массива целиком и конкатенация (сцепление) двух массивов. Кроме того, для любого значения *типа массива* можно узнать значение его **CS**.

Типы мультимножеств

При определении столбца таблицы *типа мультимножеств* используется конструкция **dt MULTISSET**, где **dt** задает тип данных элементов конструируемого *типа мультимножеств*. Значениями *типа мультимножеств* являются мультимножества, т. е. неупорядоченные коллекции элементов одного и того же типа, среди которых допускаются дубликаты. Например, значениями типа **INTEGER MULTISSET** являются мультимножества, элементами которых — целые числа. Примером такого значения может быть мультимножество **{12, 34, 12, 45, -64}**.

В отличие от массива, мультимножество является неограниченной коллекцией; при конструировании *типа мультимножеств* не указывается предельная кардинальность значений этого типа. Однако это не означает, что возможность вставки элементов в мультимножество действительно не ограничена; стандарт всего лишь не требует явного объявления границы. Ситуация аналогична той, которая возникает при работе с таблицами, для которых в **SQL** не объявляется максимально допустимое число строк.¹⁸

Для *типов мультимножеств* поддерживаются операции преобразования *типа* значения-мультимножества к *типу массивов* или другому *типу мультимножеств* с совместимым *типом* элементов (операция **CAST**), для удаления дубликатов из мультимножества (функция **SET**), для определения числа элементов в заданном мультимножестве (функция **CARDINALITY**), для выборки элемента мультимножества, содержащего в точности один элемент (функция **ELEMENT**). Кроме того, для мультимножеств обеспечиваются операции объединения (**MULTISSET UNION**), пересечения (**MULTISSET INTERSECT**) и определения разности (**MULTISSET EXCEPT**). Каждая из операций может выполняться в режиме с сохранением дубликатов (режим **ALL**) или с устранением дубликатов (режим **DISTINCT**).

Расширенные в **SQL:2003** возможности работы с *типами коллекций* являются принципиально важными. Даже при наличии определяемых пользователями *типов данных* (см. ниже) и *типов массивов SQL:1999* не предоставлял полных возможностей для преодоления исторически присущего реляционной модели данных вообще и **SQL** в частности ограничения "плоских таблиц". После появления конструктора *типов мультимножеств* и устранения ограничений на тип данных элементов коллекции это историческое ограничение полностью ликвидировано. Мультимножество, *типом* элементов которого является анонимный строчный *тип* (см. ниже), представляет собой полный аналог таблицы. Тем самым, в базе данных допускается произвольная вложенность таблиц. Возможности выбора структуры базы данных безгранично расширяются.

Анонимные строчные типы

Анонимный строчный *тип*¹⁹ — это конструктор *типов ROW*, позволяющий производить безымянные типы строк (кортежей). Любой возможный строчный *тип* получается путем использования конструктора **ROW**. При определении столбца, значения которого должны принадлежать некоторому строчному *типу*, используется конструкция **ROW (fld1, fld2, s, fldn)**, где каждый элемент **fldi**, определяющий поле

строчного типа, задается в виде тройки **fldname**, **fldtype**, **fldoptions**. Подэлемент **fldname** задает имя соответствующего поля строчного типа. Подэлемент **fldtype** специфицирует тип данных этого поля. В качестве типа данных поля строчного типа можно использовать любой допустимый в SQL тип данных, включая *типы коллекций*, *определяемые пользователями типы* и другие строчные типы. Необязательный подэлемент **fldoptions** может задаваться для указания применяемого по умолчанию порядка сортировки, если соответствующий подэлемент **fldtype** указывает на *тип символьных строк*, а также должен задаваться, если **fldtype** указывает на *ссылочный тип* (см. ниже). Степенью строчного типа называется число его полей.

Типы, определяемые пользователем

Эта категория типов данных связана с объектными расширениями языка SQL. Более подробно мы обсудим эту тему в заключительной лекции курса, а здесь для полноты картины приведем беглый набросок.

- *Структурные типы (Structured Types)*. Соответствующие возможности SQL:1999 позволяют определять долговременно хранимые, именованные типы данных²⁰, включающие один или более атрибутов любого из допустимых в SQL типа данных²¹, в том числе другие *структурные типы*, *типы коллекций*, строчные типы и т. д. Стандарт SQL не накладывает ограничений на сложность получаемой в результате структуры данных, однако не запрещает устанавливать такие ограничения в реализации. Дополнительные механизмы определяемых пользователями методов, функций и процедур позволяют определить поведенческие аспекты *структурного типа*.
- *Индивидуальные типы (Distinct Types)*. Можно определить долговременно хранимый, именованный тип данных, опираясь на единственный предопределенный тип. Например, можно определить *индивидуальный тип* данных **PRICE**, опираясь на тип **DECIMAL (5, 2)**. Тогда значения типа **PRICE** представляются точно так же, как значения типа **DECIMAL (5, 2)**. Однако в SQL:1999 *индивидуальный тип* не наследует от своего опорного типа набор операций над значениями. Например, чтобы сложить два значения типа **PRICE** требуется явно сообщить системе, что с этими значениями нужно обращаться как со значениями типа **DECIMAL (5, 2)**. Другая возможность состоит в явном определении методов, функций и процедур, связанных с данным *индивидуальным типом*. Похоже, что в будущих версиях стандарта появятся и другие, более удобные возможности.

Ссылочные типы

Эта категория типов данных связана с объектными расширениями языка SQL, и мы снова отложим подробное обсуждение этого механизма до заключительной лекции курса и рассмотрим его здесь очень коротко. Обеспечивается механизм конструирования типов (*ссылочных типов*), которые могут использоваться в качестве типов столбцов некоторого вида таблиц (*типизированных таблиц*). Фактически значениями *ссылочного типа* являются строки соответствующей *типизированной таблицы*. Более точно, каждой строке *типизированной таблицы* приписывается уникальное значение (нечто вроде *первичного ключа*, назначаемого системой или приложением), которое может использоваться в методах, определенных для табличного типа, для уникальной идентификации строк соответствующей таблицы. Эти уникальные значения называются *ссылочными значениями*, а их тип – *ссылочным типом*. *Ссылочный тип* может содержать только те значения, которые действительно ссылаются на экземпляры указанного типа (т. е. на строки соответствующей *типизированной таблицы*).

Средства определения, изменения определения и отмены определения доменов

Как неоднократно упоминалось выше, при определении столбцов таблицы требуется явно указывать *тип данных* каждого столбца. Для этого можно использовать описанные выше средства *спецификации типа*. Но в SQL поддерживается и другой механизм— механизм доменов. *Домен* является долговременно хранимым, именованным объектом *схемы базы данных*. Домены можно создавать (определять), изменять (изменять определения) и ликвидировать (отменять *определение*). Имена доменов можно использовать при определении столбцов таблиц. Можно считать, что в SQL *определение домена* представляет собой вынесенное за пределы определения индивидуальной таблицы "родовое" *определение столбца*, которое можно использовать для определения различных реальных столбцов реальных базовых таблиц. В языке SQL обеспечиваются средства *определения доменов, изменения и отмены существующих определений*.

Определение домена

Для *определения домена* в SQL используется оператор **CREATE DOMAIN**. Общий синтаксис этого оператора следующий: ²²

```
domain_definition ::= CREATE DOMAIN domain_name [AS] data_type  
    [ default_definition ]  
    [ domain_constraint_definition_list ]
```

Здесь **domain_name** задает имя создаваемого домена ²³, **data_type** есть спецификация определяющего типа данных. В необязательных разделах **default_definition** и **domain_constraint_definition_list** специфицируются *значение домена по умолчанию* ²⁴ и набор *ограничений целостности*, которые будут применяться к любому столбцу, определенному на этом домене.

Раздел **default_definition** имеет вид

```
DEFAULT { literal | niladic_function | NULL } 25
```

Здесь **literal** представляет любое допустимое литеральное значение определяющего типа домена, **NULL** обозначает *неопределенное значение*, а **niladic_function** может задаваться в одной из следующих форм:

USER

CURRENT_USER

SESSION_USER

SYSTEM_USER

CURRENT_DATE

CURRENT_TIME

CURRENT_TIMESTAMP ²⁶

Если в операторе **CREATE DOMAIN** значение по умолчанию не специфицируется, считается, что такого значения нет. Однако позже к *определению домена* можно добавить

раздел значения по умолчанию с помощью оператора **ALTER DOMAIN**. Кроме того, этот оператор позволяет удалить раздел значения по умолчанию из существующего *определения домена*.

Элемент списка **domain_constraint_definition_list** имеет вид

```
[CONSTRAINT constraint_name]
  CHECK (conditional_expression)
```

Необязательный раздел **CONSTRAINT constraint_name** позволяет определить имя нового ограничения целостности. Если явное указание имени отсутствует, ограничению назначается имя, автоматически генерируемое системой. Что касается вида условного выражения, служащего собственно *ограничением целостности*, то в стандарте запрещается лишь прямое или косвенное использование в нем домена, *вопределение* которого входит данное *условное выражение*.²⁷ Однако наиболее естественным (и наиболее распространенным) видом *ограничения домена* является следующий:

```
CHECK (VALUE IN (list_of_valid_values))
```

Такое ограничение запрещает появление в любом столбце, определенном на данном домене, любого значения определяющего типа, не входящего в список допустимых значений.

Примеры определений доменов

В дальнейших примерах нам понадобятся *определения* нескольких доменов. Приведем их в этом подразделе. В примерах мы будем иметь дело с таблицами служащих (**EMP**), отделов (**DEPT**) и проектов (**PRO**). Каждый служащий обладает уникальным номером (**EMP_NO**) и получает *заработную плату* (**SALARY**). Определим домены **EMP_NO** и **SALARY**.

```
CREATE DOMAIN EMP_NO AS INTEGER
  CHECK (VALUE BETWEEN 1 AND 10000);
```

Номера служащих являются целыми числами, поэтому *базовый тип* домена **EMP_NO** есть тип **INTEGER**. Кроме того, на значения этого домена устанавливается следующее ограничение: они должны быть больше нуля и не превосходить целое значение 10000.

Домен **SALARY** определим следующим образом:

```
CREATE DOMAIN SALARY AS NUMERIC (10, 2)
  DEFAULT 10000.00
  CHECK (VALUE BETWEEN 10000.00 AND 20000000.00)
  CONSTRAINT SAL_NOT_NULL CHECK (VALUE IS NOT NULL);
```

Размер *заработной платы* является значением *точного числового типа* **NUMERIC** из десяти десятичных цифр, две из которых составляют дробную часть. По умолчанию размер *заработной платы* составляет 10000 руб. Установлен диапазон допустимого размера зарплаты от 10000 руб. до 20000000 руб. *Неопределенное значение* зарплаты не допускается (на уровне *определения домена*).

Изменение определения домена

Для *изменения* характеристик ранее определенного домена используется оператор SQL **ALTER DOMAIN**. Синтаксис этого оператора выглядит следующим образом:

```
domain_alteration ::=
    ALTER DOMAIN domain_name domain_alteration_action
domain_alteration_action ::=
    domain_default_alteration_action
    | domain_constraint_alteration_action
```

Как видно из синтаксических правил, при *изменении определения домена* можно выполнить действие по изменению раздела значения по умолчанию либо изменить *ограничение домена*. Для первого варианта действует следующий синтаксис:

```
domain_default_alteration_action ::=
    SET default_definition
    | DROP DEFAULT
```

В случае установки нового значения по умолчанию (**SET**) это значение автоматически применяется ко всем столбцам, определенным на данном домене. Более точно, это значение становится новым значением по умолчанию. Операция не оказывает влияния на состояние существующих строк таблиц базы данных. В случае отмены раздела *значения по умолчанию в определении домена* (**DROP**) существовавшее *значение домена по умолчанию* становится значением по умолчанию каждого столбца, который определен на данном домене и для которого не специфицировано собственное значение по умолчанию.

Действие по изменению *ограничения домена* определяется следующим синтаксисом:

```
domain_constraint_alteration_action ::=
    ADD domain_constraint_definition
    | DROP CONSTRAINT constraint_name
```

Действие по добавлению нового определения *ограничения домена* (**ADD**) приводит к тому, что новое условие добавляется через **AND** к существующему *ограничению домена*. Если к моменту выполнения соответствующего оператора **ALTER DOMAIN** существуют столбцы некоторых таблиц, текущие значения которых противоречат новому ограничению, то СУБД должна отвергнуть этот оператор **ALTER DOMAIN**. Действие по отмене *ограничения домена* (**DROP**) приводит к исчезновению соответствующей части общего ограничения соответствующего домена, что, естественно, не влияет на существующие значения столбцов имеющихся таблиц.

Примеры изменения определения домена

Немного поупражняемся с доменом **SALARY**. Для изменения значения *заработной платы* по умолчанию с 10000 на 11000 руб. нужно выполнить оператор

```
ALTER DOMAIN SALARY SET DEFAULT 11000.00;
```

Для отмены значения по умолчанию в домене **SALARY** следует воспользоваться оператором

ALTER DOMAIN SALARY DROP DEFAULT;

Если к определению домена **SALARY** требуется добавить ограничение (например, запретить значение зарплаты, равное 15000 руб.), необходимо выполнить оператор

ALTER DOMAIN SALARY ADD CHECK (VALUE <> 15000.00);

Наконец, если требуется отменить (именованное!) *ограничение целостности*, препятствующее наличию неопределенных значений в столбцах, которые определены на домене **SALARY**, то нужно выполнить оператор

ALTER DOMAIN SALARY DROP CONSTRAINT SAL_NOT_NULL;

Отмена определения домена

Чтобы отменить ранее созданное *определение домена*, нужно воспользоваться оператором **DROP DOMAIN** в следующем синтаксисе:

DROP DOMAIN domain_name {RESTRICT | CASCADES}

Если в операторе указано **RESTRICT**, и если соответствующий домен использован в определении некоторого столбца, в определении некоторого представления или в определении ограничения целостности (см. следующие лекции), то оператор **DROP DOMAIN** отвергается. В противном случае *определение домена* ликвидируется.

Если в операторе **DROP DOMAIN** указано **CASCADES**, то оператор выполняется всегда. При этом уничтожаются все представления и ограничения целостности, в определении которых использовалось имя данного домена. Столбцы, определенные на этом домене, автоматически переопределяются следующим образом:

- считается, что каждый такой столбец теперь относится к определяющему типу уничтожаемого домена;
- если у столбца не было определено собственное значение по умолчанию, то считается, что теперь у него имеется такое значение по умолчанию, совпадающее со значением по умолчанию уничтожаемого домена;
- каждый столбец наследует все *ограничения* уничтожаемого домена.

Неявные и явные преобразования типа или домена

В языке **SQL** обеспечивается возможность использования в различных операциях не только значений тех типов, для которых предопределена операция, но и значений типов, неявным или явным образом приводимых к требуемому типу.

Неявные преобразования типов в SQL

В **SQL** поддерживается совместимость некоторых типов данных за счет *неявного преобразования* значений одного типа к значениям другого типа данных (например, при необходимости **FLOAT** неявно приводится к **DOUBLE**). Опишем наиболее важные правила *совместимости типов*, принятые в **SQL:1999**. Начнем с определения приводимости типов. Тип данных **A** приводим к типу данных **B** в том и только в том

случае, когда в любом месте, где ожидается значение типа **B**, может быть использовано значение типа **A**.

Основные правила приводимости типов состоят в следующем.

- *Типы символьных строк.* Тип **CHARACTER** (x) приводим к любому типу **CHARACTER** (y), если $y \geq x$. Типы **VARCHAR** (x) и **CHARACTER** (x) приводимы к любому типу **VARCHAR** (y), если $y \geq x$. Типы **CHARACTER** (x) и **VARCHAR** (x) приводимы к любому типу **CLOB**.
- *Типы битовых строк.* Тип **BIT** (x) приводим к любому типу **BIT** (y), если $y \geq x$. Типы **BIT VARYING** (x) и **BIT** (x) приводимы к любому типу **BIT VARYING** (y), если $y \geq x$.
- *Типы BLOB.* Тип **BLOB** (x) приводим к любому типу **BLOB** (y), если $y \geq x$.
- *Типы точных чисел.* Тип **EN** (p₁, s₁) приводим к любому типу **EN** (p₂, s₂), у которого $s_2 \geq s_1$ и p₂ определяется в реализации. Тип **EN** (p, s) приводим к любому типу приближенных чисел **AN** (p₁), где p₁ определяется в реализации.
- *Типы приближенных чисел.* Тип **AN** (p₁) приводим к любому типу (p₂), если $p_2 \geq p_1$.

Явные преобразования типов или доменов и оператор CAST

Неявные преобразования типов не всегда удобны, недостаточно гибки и иногда могут вызывать ошибки. Поэтому, как показывает предыдущий подраздел, число допустимых неявных преобразований типов в SQL весьма ограничено. Однако в SQL существует специальный оператор **CAST**, с помощью которого можно *явно преобразовывать типы или домены* в более широких пределах допускаемых преобразований. Конструкция имеет следующий синтаксис:

CAST ({scalar-expression | NULL } AS
{data_type | domain_name})

Оператор преобразует значение заданного *скалярного выражения* к указанному типу или к базовому типу указанного домена. Результатом применения оператора **CAST** к *неопределенному значению* является *неопределенное значение*. Для значений, отличных от неопределенных, в стандарте приводятся подробные правила выполнения преобразований, которые интуитивно понятны.

Поясним действие оператора **CAST** в наиболее важных случаях. Примем следующие обозначения типов данных:

EN – точные числовые типы (*Exact Numeric*)

AN – приближенные числовые типы (*Approximate Numeric*)

C – типы символьных строк (*Character*)

FC – типы символьных строк постоянной длины (*Fixed-length Character*)

VC – типы символьных строк переменной длины (*Variable-length Character*)

B – типы битовых строк (*Bit String*)

FB – типы битовых строк постоянной длины (*Fixed-length Bit String*)

VB – типы битовых строк переменной длины (*Variable-length Bit String*)

D – тип *Date*

T – типы *Time*

TS – типы *Timestamp*

YM – типы *Interval Year-Month*

DT – типы *Interval Day-Time*

Пусть **TD** – это тип данных, к которому производится преобразование, а **SD** – тип данных операнда. Тогда допустимы следующие комбинации ("да" означает безусловную допустимость, "нет" – безусловную недопустимость и "?" – допустимость с оговорками).

TD												
	EN	AN	VC	FC	VB	FB	D	T	TS	YM	DT	
EN	Да	Да	Да	Да	Нет	Нет	Нет	Нет	Нет	?	?	
AN	Да	Да	Да	Да	Нет	Нет	Нет	Нет	Нет	Нет	Нет	
C	Да	Да	?	?	Да	Да	Да	Да	Да	Да	Да	
SD B	Нет	Нет	Да	Да	Да	Да	Нет	Нет	Нет	Нет	Нет	
D	Нет	Нет	Да	Да	Нет	Нет	Да	Нет	Да	Нет	Нет	
T	Нет	Нет	Да	Да	Нет	Нет	Нет	Да	Да	Нет	Нет	
TS	Нет	Нет	Да	Да	Нет	Нет	Да	Да	Да	Нет	Нет	
YM	?	Нет	Да	Да	Нет	Нет	Нет	Нет	Нет	Да	Нет	
DT	?	Нет	Да	Да	Нет	Нет	Нет	Нет	Нет	Нет	Да	

По поводу ячеек таблицы, содержащих знак вопроса, необходимо сделать несколько оговорок:

1. если **TD** – интервал и **SD** – тип точных чисел, то **TD** должен содержать единственное поле даты-времени;
2. если **TD** – тип точных чисел и **SD** – интервал, то **SD** должен содержать единственное поле даты-времени;
3. если **SD** – тип символьных строк и **TD** – тип символьных строк постоянной или переменной длины, то набор символов **SD** и **TD** должен быть одним и тем же.

Заключение

В этой лекции мы начали рассматривать средства языка *SQL*, позволяющие определять и динамически изменять схему *базы данных*. Наиболее важным для общего понимания языка является раздел "Типы данных *SQL*" – *система типов языка SQL* (и любой *SQL* - ориентированной *базы данных*). В последних стандартах языка *SQL* поддерживаются:

- развитый набор предопределенных типов, включая ряд параметризованных типов;
- генераторы *типов массивов* и *мультимножеств*, элементами которых могут быть значения предопределенных типов, *типов коллекций*, *анонимных строчных типов* строк и типов, определенных пользователями;
- генератор *анонимных строчных типов*, в которых типом элемента строки может быть любой предопределенный тип, *тип коллекции*, *анонимный строчный тип* и тип, определенный пользователями;
- *определяемый пользователем структурный тип*, в котором типом элемента структуры может быть любой предопределенный тип, *тип коллекции*, *анонимный строчный тип* и тип, определенный пользователями; для определяемых пользователем *структурных* и *индивидуальных типов* можно определять пользовательские операции.

Нельзя с уверенностью сказать, что *система типов языка SQL* настолько полна, что может удовлетворить любые потребности, но можно отметить, что в этой *системе типов* отсутствует единый *логический* подход и имеется *избыточность*. Возможно, это станет понятнее после обсуждения в конце курса средств объектно-реляционных расширений языка *SQL*.

Как должно быть ясно из этой лекции, механизм доменов в *SQL* играет вспомогательную роль. Это не совсем те (может быть, и совсем не те) домены, *поддержка* которых предполагается реляционной моделью. Фактически *определение домена* обеспечивает спецификацию ограничений и значений по умолчанию, выносимых за пределы определения столбца. В комитете по стандартизации *SQL* обсуждается идея полного отказа от поддержки механизма доменов и замены его на соответствующим образом адаптированный механизм *индивидуальных типов* (см. последнюю лекцию курса).

Лабораторная работа2. Средства определения базовых таблиц и ограничений целостности

Введение

Как мы уже отмечали ранее, к спецификации языка *SQL* можно относиться как к спецификации некоторой *модели данных*, в определенных аспектах близкой к реляционной модели. Мы стремимся к тому, чтобы порядок лекций, посвященных языку *SQL*, способствовал правильному пониманию именно этой модели, а не технических тонкостей языка. Предыдущая Лабораторная работа посвящалась тому, что (т. е. данные каких типов) может храниться в *SQL*-ориентированной базе данных.

Теперь следует понять, где хранятся эти данные. Как и в *реляционной модели данных*, в модели SQL поддерживается единственная родовая структура данных, называемая в данном случае *базовой таблицей*. В первом из двух основных разделов лекции обсуждаются средства языка SQL, предназначенные для определения, изменения определения и отмены определения *базовых таблиц*.

Понятие *базовой таблицы* родственно понятию отношения: можно считать, что *базовая таблица* обладает заголовком¹, в котором содержатся различаемые имена столбцов и их типы данных (заголовок *базовой таблицы* является множеством и представляет собой близкий аналог заголовка отношения), и телом, включающим строки, которые соответствуют заголовку таблицы (казалось бы, здесь мы имеем аналоги *тела отношения* и *кортежей*). Но коренное отличие *базовой таблицы* от истинного отношения состоит в том, что тело таблицы не обязательно является множеством. Среди строк тела таблицы могут встречаться дубликаты, и в общем случае тело *базовой таблицы SQL* представляет собой *мультимножество строк*.

Забегая вперед (см. следующие лекции), следует заметить, что порождаемые таблицы SQL, которые формируются при выполнении запросов к SQL-ориентированной базе данных, еще более отдаляют SQL от реляционной модели. В таких таблицах может отсутствовать и правильно сформированный заголовок (могут иметься одноименные столбцы).

Почему же, понимая принципиальные отклонения языка SQL от *реляционной модели данных*, мы включили эти две темы в один курс и, более того, иногда неформально называем SQL языком *реляционных баз данных*? Тому есть несколько причин.

- Во-первых, используя язык SQL, можно не нарушать предписаний реляционной модели, и тогда к "правильно построенной" SQL-ориентированной базе данных применимы все фундаментальные результаты *теории реляционных баз данных*, включая принципы проектирования на основе *нормализации*.
- Во-вторых, полностью отвергая родство языка SQL с реляционной моделью данных, мы выступали бы против установившихся исторических традиций. Этот язык возник около 30 лет тому назад во время реализации в компании IBM проекта по созданию экспериментальной СУБД System R, основной целью которого являлось обоснование практической реализации *реляционного подхода к организации баз данных*. Так что исторически SQL базировался на *реляционной модели данных* (возможно, не совсем верно понятой и/или воплощенной).
- Наконец, по нашему мнению, в области информационной технологии любой практически используемый инструмент не может быть полностью свободен от компромиссов. Идеологически чистые решения возможны только в научно-экспериментальной работе. "Великий и ужасный" язык SQL – это порождение ряда компромиссов между теорией, практикой и маркетинговой деятельностью. Этот язык является настолько реляционным, насколько это понадобилось потребителям коммерческих СУБД, прямо или косвенно финансировавшим разработку языка.

В операторе SQL **CREATE TABLE** специфицируются не только столбцы таблицы, но и *ограничения целостности*, которым должны удовлетворять данные, хранящиеся в *базовой таблице*. Эти ограничения являются частным случаем ограничений *базы данных* целиком, для определения которых, а также изменения и ликвидации определений имеются специальные *операторы*. Обсуждению этих средств посвящен второй *основной раздел* этой лекции.

Средства определения, изменения и ликвидации базовых таблиц

Базовые (реально хранимые в базе данных) таблицы создаются (определяются) с использованием оператора **CREATE TABLE**. Для изменения определения базовой таблицы применяется оператор **ALTER TABLE**. Уничтожить хранимую таблицу (отменить ее определение) можно с помощью оператора **DROP TABLE**.

Замечание: хотя внешне операторы **CREATE TABLE**, **ALTER TABLE** и **DROP TABLE** похожи на соответствующие операторы определения, изменения определения и отмены определения домена, между ними имеется принципиальное различие. Определение домена приводит всего лишь к созданию некоторых новых описателей, входящих в состав метаданных базы данных. Создание базовой таблицы, кроме создания соответствующих описателей, порождает новую область внешней памяти, в которой будут храниться данные, поставляемые пользователями. Тем самым, базовая таблица SQL-ориентированной базы данных является прямым аналогом переменной отношения реляционной модели данных.

Определение базовой таблицы

Оператор создания базовой таблицы **CREATE TABLE** имеет следующий синтаксис:

base_table_definition ::= CREATE TABLE base_table_name (base_table_element_commlist)²

base_table_element ::= column_definition | base_table_constraint_definition

Здесь **base_table_name** задает имя новой (изначально пустой) базовой таблицы. Каждый элемент определения базовой таблицы является либо определением столбца, либо определением табличного ограничения целостности.

Определение столбца

Элемент определения столбца специфицируется на основе следующих синтаксических правил:

column_definition ::= column_name
{ data_type | domain_name }
[default_definition]
[column_constraint_definition_list]

В элементе определения столбца **column_name** задает имя определяемого столбца. Тип столбца специфицируется путем явного указания типа данных (**data_type**) или путем указания имени ранее определенного домена (**domain_name**).

Значения столбца по умолчанию

Необязательный раздел определения значения столбца по умолчанию имеет тот же синтаксис, что и раздел определения значения по умолчанию в операторах определения или изменения определения домена:

DEFAULT { literal | niladic_function | NULL }

Действующее значение по умолчанию для данного столбца определяется следующим образом:

- если в определении столбца явно присутствует раздел **DEFAULT**, то значением столбца по умолчанию является значение, указанное в этом разделе;
- иначе, если столбец определяется на домене и в определении этого домена явно присутствует раздел **DEFAULT**, то значением столбца по умолчанию является значение, указанное в этом разделе;
- иначе значением по умолчанию столбца является **NULL**.³

Заметим, что если значением по умолчанию неявно объявлено неопределенное значение (**NULL**), но среди ограничений целостности столбца присутствует ограничение **NOT NULL** (см. ниже), то считается, что у столбца вообще отсутствует значение по умолчанию. Это означает, что при любой вставке новой строки в соответствующую базовую таблицу значение данного столбца должно быть задано явно.

Ограничения целостности столбца

Элемент необязательного списка ограничений целостности столбца определяется следующими синтаксическими правилами:

```
column_constraint_definition ::=  
    [ CONSTRAINT constraint_name ]  
    NOT NULL  
    | { PRIMARY KEY | UNIQUE }  
    | references_definition  
    | CHECK ( conditional_expression )
```

Как мы увидим немного позже, любое ограничение целостности, включаемое в определение столбца, может быть эквивалентным образом выражено в виде табличного ограничения целостности. Единственный резон определения ограничений на уровне столбца состоит в том, что в этом случае в ограничении целостности не требуется явно указывать имя столбца. Тем не менее, кратко рассмотрим ограничения целостности столбца отдельно.

Ограничение **NOT NULL** означает, что в определяемом столбце никогда не должны содержаться неопределенные значения. Если определяемый столбец имеет имя **C**, то это ограничение эквивалентно следующему табличному ограничению: **CHECK (C IS NOT NULL)**.

В определение столбца может входить одно из ограничений уникальности: ограничение первичного ключа (**PRIMARY KEY**) или ограничение возможного ключа (**UNIQUE**). Включение в определение столбца любого из этих ограничений означает требование уникальности значений определяемого столбца (т. е. во все время существования определяемой таблицы во всех ее строках значения данного столбца должны быть различны⁴). Ограничение **PRIMARY KEY**, в дополнение к этому, влечет за собой ограничение **NOT NULL** для определяемого столбца. Эти ограничения столбца эквивалентны следующим табличным ограничениям: **PRIMARYKEY (C)** и **UNIQUE (C)**.

Ограничение **references_definition** означает объявление определяемого столбца внешним ключом таблицы и обладает следующим синтаксисом:

references_definition ::=

```
REFERENCES base_table_name [ (column_commalist) ]  
  [ MATCH { SIMPLE | FULL | PARTIAL } ]  
  [ ON DELETE referential_action ]  
  [ ON UPDATE referential_action ]
```

На самом деле, данная синтаксическая конструкция работает и в случае *определения внешнего ключа на уровне таблицы* (в одном из *определений табличных ограничений* целостности). Поэтому мы отложим обсуждение до рассмотрения этого общего случая. Пока отметим только, что при использовании конструкции на уровне *определения столбца* **column_commalist** может содержать имя только одного столбца (потому что внешний ключ состоит из одного определяемого столбца). Ограничение эквивалентно следующему табличному ограничению: **FOREIGN KEY (C) references_definition**.

Проверочное ограничение CHECK (conditional_expression) приводит к тому, что в данном столбце могут находиться только те значения, для которых вычисление **conditional_expression** не приводит к результату **false**. В условном выражении *проверочного ограничения столбца* разрешается использовать имя только определяемого столбца. Заметим, что *проверочное ограничение столбца* может быть безо всяких изменений перенесено на уровень *определения табличных ограничений*.

Определение табличного ограничения

Элемент *определения табличного ограничения* целостности задается в следующем синтаксисе:

base_table_constraint_definition ::=

```
[ CONSTRAINT constraint_name ]  
{ PRIMARY KEY | UNIQUE } ( column_commalist )  
| FOREIGN KEY ( column_commalist )  
  references_definition  
| CHECK ( conditional_expression )
```

Как мы видим, имеется три разновидности табличных ограничений: *ограничение первичного* или *возможного ключа* (**PRIMARY KEY** или **UNIQUE**), *ограничение внешнего ключа* (**FOREIGN KEY**) и *проверочное ограничение* (**CHECK**). Любому ограничению может явным образом назначаться имя, если перед определением ограничения поместить конструкцию **CONSTRAINT constraint_name**.

Табличное ограничение первичного или возможного ключа

Табличное ограничение первичного или возможного ключа { **PRIMARY_KEY** | **UNIQUE** } (column_commalist) означает требование уникальности составных значений указанной группы столбцов (т. е. во все время существования определяемой таблицы во всех ее строках составные значения данной группы столбцов должны быть различны⁵). Ограничение **PRIMARY KEY**, в дополнение к этому, влечет ограничение **NOT NULL** для всех столбцов, упоминаемых в определении ограничения. В определении таблицы допускается произвольное число определений *возможного ключа* (для разных комбинаций столбцов), но не более одного определения *первичного ключа*. Обратите особое внимание на последнюю часть предыдущего предложения: в языке SQL

действительно допускается определение таблиц, у которых отсутствуют возможные ключи. Эта особенность языка, среди прочего, очевидным образом противоречит базовым требованиям *реляционной модели данных*.

Проверочное табличное ограничение

Определение табличного ограничения вида **CHECK (conditional_expression)** приводит к тому, что указанное *условное выражение* будет вычисляться при каждой попытке обновления соответствующей таблицы (вставке новой строки, удалении или модификации существующей строки). Считается, что попытка обновления таблицы нарушает *проверочное ограничение* целостности, если после выполнения операции обновления вычисление *условного выражения* дает результат **false**. Другими словами, таблица находится в соответствии с данным *проверочным табличным ограничением*, если для всех строк таблицы результатом вычисления соответствующего *условного выражения* не является **false**.

Мы отложим обсуждение допустимых разновидностей *условных выражений* до следующей лекции, где оно будет более уместно в контексте рассмотрения оператора **SELECT** языка SQL.

Табличное ограничение внешнего ключа

Синтаксис и семантика определения *внешнего ключа* в операторе SQL определения *базовой таблицы* являются довольно запутанными и сложными. По этой причине мы посвящаем этой языковой конструкции отдельный подраздел.

Табличное ограничение **FOREIGN KEY (column_commalist) references_definition** означает объявление *внешним ключом* группы столбцов, имена которых перечислены в списке **column_commalist**. Обсудим теперь смысл *ограничения внешнего ключа* при разных вариантах формирования определения ссылок (**references_definition**). Для удобства повторим синтаксическое правило.

references_definition ::=

```
REFERENCES base_table_name [ (column_commalist) ]  
    [ MATCH { SIMPLE | FULL | PARTIAL } ]  
    [ ON DELETE referential_action ]  
    [ ON UPDATE referential_action ]
```

В этом определении **base_table_name** должно представлять собой имя некоторой *базовой таблицы* (пусть, например, эта таблица имеет имя **T**). Если определение ссылок включает список столбцов (**column_commalist**), то этот список должен совпадать (с точностью до порядка следования имен столбцов) со списком имен столбцов, использованных в некотором определении *первичного* или *возможного* *ключа* (**PRIMARY_KEY** или **UNIQUE**) в определении таблицы **T**. Если в определении ссылок список столбцов явно не задан, то считается, что он совпадает со списком столбцов, использованных в определении *первичного* *ключа* (**PRIMARY_KEY**) таблицы **T**.

Разновидности способов сопоставления значений внешнего и возможного ключей

Пусть определяемая таблица имеет имя **S**. Обсудим смысл необязательного раздела определения внешнего ключа **MATCH { SIMPLE | FULL | PARTIAL }**. Если этот раздел отсутствует или если присутствует и имеет вид **MATCH SIMPLE**, то *ограничение внешнего ключа* (ссылочное ограничение) удовлетворяется в том и только в том случае, когда для каждой строки таблицы **S** выполняется одно из следующих условий:

- (a) какой-либо столбец, входящий в состав *внешнего ключа*, содержит **NULL** ;
- (b) таблица **T** содержит в точности одну строку, такую, что значение *внешнего ключа* в данной строке таблицы **S** совпадает со значением соответствующего *возможного ключа* в этой строке таблицы **T**.

Если раздел **MATCH** присутствует в определении *внешнего ключа* и имеет вид **MATCH PARTIAL**, то *ограничение внешнего ключа* удовлетворяется в том и только в том случае, когда для каждой строки таблицы **S** выполняется одно из следующих условий:

- (a) каждый столбец, входящий в состав *внешнего ключа*, содержит **NULL** ;
- (b) таблица **T** содержит по крайней мере одну такую строку, что для каждого столбца данной строки таблицы **S**, значение которого отлично от **NULL**, его значение совпадает со значением соответствующего столбца *возможного ключа* в этой строке таблицы **T**.

Если раздел **MATCH** имеет вид **MATCH FULL**, то *ограничение внешнего ключа* удовлетворяется в том и только в том случае, когда для каждой строки таблицы **S** выполняется одно из следующих условий:

- (a) *каждый столбец*, входящий в состав *внешнего ключа*, содержит **NULL** ;
- (b) ни один столбец, входящий в состав *внешнего ключа*, не содержит **NULL**, и таблица **T** содержит в точности одну строку, такую, что значение *внешнего ключа* в данной строке таблицы **S** совпадает со значением соответствующего *возможного ключа* в этой строке таблицы **T**.

Очевидно, что только при наличии спецификации **MATCH FULL** ссылочное ограничение соответствует требованиям реляционной модели. Тем не менее в определении *ограничения внешнего ключа базовых таблиц* в SQL по умолчанию предполагается наличие спецификации **MATCH SIMPLE**⁶

Поддержка ссылочной целостности и ссылочные действия

В связи с определением *ограничения внешнего ключа* нам осталось рассмотреть еще два необязательных раздела – **ON DELETE referential_action** и **ON UPDATE referential_action**. Прежде всего, приведем синтаксическое правило:

```
referential_action ::=  
    { NO ACTION | RESTRICT | CASCADE  
    | SET DEFAULT | SET NULL }
```

Чтобы объяснить, в каких случаях и каким образом выполняются эти действия, требуется сначала определить понятие ссылающейся строки (**referencing row**). Если в определении *ограничения внешнего ключа* отсутствует раздел **MATCH** или присутствуют спецификации **MATCH SIMPLE** либо **MATCH FULL**, то для данной

строки **t** таблицы **T** строкой таблицы **S**, ссылающейся на строку **t**, называется каждая строка таблицы **S**, значение *внешнего ключа* которой совпадает со значением соответствующего *возможного ключа* строки **t**. Если в определении *ограничения внешнего ключа* присутствует спецификация **MATCHPARTIAL**, то для данной строки **t** таблицы **T** строкой таблицы **S**, ссылающейся на строку **t**, называется каждая строка таблицы **S**, отличные от **NULL** значения столбцов *внешнего ключа* которой совпадают со значениями соответствующих столбцов соответствующего *возможного ключа* строки **t**. В случае **MATCH PARTIAL** строка таблицы **S** называется ссылающейся исключительно на строку **t** таблицы **T**, если эта строка таблицы **S** является ссылающейся на строку **t** и не является ссылающейся на какую-либо другую строку таблицы **T**.⁷

Теперь приступим к *ссылочным действиям*. Пусть определение *ограничения внешнего ключа* содержит раздел **ON DELETE referential_action**. Предположим, что предпринимается попытка удалить строку **t** из таблицы **T**. Тогда:

- если в качестве требуемого *ссылочного действия* указано **NO ACTION** или **RESTRICT**, то операция удаления отвергается, если ее выполнение вызвало бы нарушение *ограничения внешнего ключа* ;
- если в качестве требуемого *ссылочного действия* указано **CASCADE**, то строка **t** удаляется, и если в определении *ограничения внешнего ключа* отсутствует раздел **MATCH** или присутствуют спецификации **MATCH SIMPLE** или **MATCH FULL**, то удаляются все строки, ссылающиеся на **t**. Если же в определении *ограничения внешнего ключа* присутствует спецификация **MATCH PARTIAL**, то удаляются только те строки, которые ссылаются исключительно на строку **t** ;
- если в качестве требуемого *ссылочного действия* указано **SET DEFAULT**, то строка **t** удаляется, и во всех столбцах, которые входят в состав *внешнего ключа*, всех строк, ссылающихся на строку **t**, проставляется заданное при их определении *значение по умолчанию*. Если в определении *внешнего ключа* содержится спецификация **MATCH PARTIAL**, то подобному воздействию подвергаются только те строки таблицы **S**, которые ссылаются исключительно на строку **t** ;
- если в качестве требуемого *ссылочного действия* указано **SET NULL**, то строка **t** удаляется, и во всех столбцах, которые входят в состав *внешнего ключа*, всех строк, ссылающихся на строку **t**, проставляется **NULL**. Если в определении *внешнего ключа* содержится спецификация **MATCH PARTIAL**, то подобному воздействию подвергаются только те строки таблицы **S**, которые ссылаются исключительно на строку **t**.

Пусть определение *ограничения внешнего ключа* содержит раздел **ON UPDATE referential_action**. Предположим, что предпринимается попытка обновить столбцы соответствующего *возможного ключа* в строке **t** из таблицы **T**. Тогда:

- если в качестве требуемого *ссылочного действия* указано **NO ACTION** или **RESTRICT**, то операция обновления отвергается, если ее выполнение вызвало бы нарушение *ограничения внешнего ключа* ;
- если в качестве требуемого *ссылочного действия* указано **CASCADE**, то строка **t** обновляется, и если в определении *ограничения внешнего ключа* отсутствует раздел **MATCH** или присутствуют спецификации **MATCH SIMPLE** или **MATCH FULL**, то соответствующим образом обновляются все строки, ссылающиеся на **t** (в них должным образом изменяются значения столбцов, входящих в состав *внешнего ключа*). Если же в определении *ограничения внешнего ключа* присутствует

спецификация **MATCH PARTIAL**, то обновляются только те строки, которые ссылаются исключительно на строку **t** ;

- если в качестве требуемого *ссылочного действия* указано **SET DEFAULT**, то строка **t** обновляется, и во всех столбцах, которые входят в состав *внешнего ключа* и соответствуют изменяемым столбцам таблицы **T**, всех строк, ссылающихся на строку **t**, проставляется заданное при их определении *значение по умолчанию*. Если в определении *внешнего ключа* содержится спецификация **MATCH PARTIAL**, то подобному воздействию подвергаются только те строки таблицы **S**, которые ссылаются исключительно на строку **t**, причем в них изменяются значения только тех столбцов, которые не содержали **NULL** ;
- если в качестве требуемого *ссылочного действия* указано **SET NULL**, то строка **t** обновляется, и во всех столбцах, которые входят в состав *внешнего ключа* и соответствуют изменяемым столбцам таблицы **T**, всех строк, ссылающихся на строку **t**, проставляется **NULL**. Если в определении *внешнего ключа* содержится спецификация **MATCH PARTIAL**, то подобному воздействию подвергаются только те строки таблицы **S**, которые ссылаются исключительно на строку **t**.⁸

Примеры определений базовых таблиц

Определим таблицы служащих (**EMP**), отделов (**DEPT**) и проектов (**PRO**). Эти таблицы имеют заголовки, показанные на [рис. 12.1](#).

EMP:
EMP_NO : EMP_NO
EMP_NAME : VARCHAR
EMP_BDATE : DATE
EMP_SAL : SALARY
DEPT_NO : DEPT_NO
PRO_NO : PRO_NO

DEPT:
DEPT_NO : DEPT_NO
DEPT_NAME : VARCHAR
DEPT_EMP_NO : INTEGER
DEPT_TOTAL_SAL : SALARY
DEPT_MNG : EMP_NO

PRO:
PRO_NO : PRO_NO
PRO_TITLE : VARCHAR
PRO_SDATE : DATE
PRO_DURAT : INTERVAL
PRO_MNG : EMP_NO
PRO_DESC : CLOB

Рис. 12.1. Заголовки таблиц EMP, DEPT и PRO

Столбцы **EMP_NO**, **EMP_SAL**, **DEPT_NO**, **PRO_NO**, **DEPT_TOTAL_SAL**, **DEPT_MNG** и **PRO_MNG** определяются на ранее определенных доменах (определения

доменов **EMP_NO** и **SALARY** приведены в предыдущей лекции). Первичными ключами отношений **EMP**, **DEPT** и проектов **PRO** являются столбцы **EMP_NO**, **DEPT_NO** и **PRO_NO** соответственно. В таблице **EMP** столбцы **DEPT_NO** и **PRO_NO** являются внешними ключами, указывающими на отдел, в котором работает служащий, и на выполняемый им проект соответственно. В таблице **DEPT** внешним ключом является столбец **DEPT_MNG**, указывающий на служащего, являющегося руководителем соответствующего отдела, а в таблице **PRO** внешним ключом является столбец **PRO_MNG**, указывающий на служащего, являющегося менеджером соответствующего проекта. Другие ограничения целостности мы обсудим позже.

Определим таблицу **EMP**:

```
(1) CREATE TABLE EMP (
(2) EMP_NO EMP_NO PRIMARY KEY,
(3) EMP_NAME VARCHAR(20) DEFAULT 'Incognito' NOT NULL,
(4) EMP_BDATE DATE DEFAULT NULL CHECK (
    VALUE >= DATE '1917-10-24'),
(5) EMP_SAL SALARY,
(6) DEPT_NO DEPT_NO DEFAULT NULL REFERENCES
    DEPT ON DELETE SET NULL,
(7) PRO_NO PRO_NO DEFAULT NULL,
(8) FOREIGN KEY PRO_NO REFERENCES PRO (PRO_NO)
    ON DELETE SET NULL,
(9) CONSTRAINT PRO_EMP_NO CHECK
    ((SELECT COUNT (*) FROM EMP E
      WHERE E.PRO_NO = PRO_NO) <= 50));
```

Последовательно обсудим части этого определения. В части (1) указывается, что создается таблица с именем **EMP**. В части (2) определяется столбец **EMP_NO** на домене **EMP_NO**. У этого столбца не определено значение по умолчанию, и он объявлен первичным ключом таблицы (это ограничение целостности добавляется через **AND** к ограничениям, унаследованным столбцом от определения домена). Помимо прочего, это означает неявное указание запрета для данного столбца неопределенных значений. В части (3) определен столбец **EMP_NAME** на базовом типе данных символьных строк переменной длины с максимальной длиной 20. Для столбца указано значение по умолчанию – строка 'Incognito', и в качестве ограничения целостности запрещены неопределенные значения. В части (4) определяется столбец **EMP_BDATE** (дата рождения служащего). Он имеет тип данных **DATE**, значением по умолчанию является **NULL** (даты рождения некоторых служащих неизвестны). Кроме того, ограничение столбца запрещает принимать на работу лиц, о которых известно, что они родились до Октябрьского переворота. В части (5) определен столбец **EMP_SAL** на домене **SALARY**. Значение по умолчанию и ограничения целостности наследуются из определения домена. В части (6) столбец **DEPT_NO** определяется на одноименном домене (для наших целей его определение несущественно), но явно объявляется, что значением по умолчанию этого столбца будет **NULL** (некоторые служащие не приписаны ни к какому отделу). Кроме того, добавляется ограничение внешнего ключа: столбец **DEPT_NO** ссылается на первичный ключ таблицы **DEPT**. Определено ссылочное действие: при удалении строки из таблицы **DEPT** во всех строках таблицы **EMP**, ссылавшихся на эту строку,

столбцу **DEPT_NO** должно быть присвоено неопределенное значение. В части (7) определяется столбец **PRO_NO**. Его определение аналогично определению столбца **DEPT_NO**, но ограничение внешнего ключа вынесено в часть (8), где оно определяется в полной форме как табличное ограничение. Наконец, в части (9) определяется табличное проверочное ограничение с именем **PRO_EMP_NO**, которое требует, чтобы ни в одном проекте не участвовало больше 50 служащих (правила построения соответствующего условного выражения поясняются в следующих лекциях).

Определим таблицу **DEPT**:

- (1) CREATE TABLE DEPT (
- (2) DEPT_NO DEPT_NO PRIMARY KEY,
- (3) DEPT_EMP_NO INTEGER NOT NULL CHECK (
VALUE BETWEEN 1 AND 100),
- (4) DEPT_NAME VARCHAR(200) DEFAULT 'Nameless' NOT NULL,
- (5) DEPT_TOTAL_SAL SALARY DEFAULT 1000000.00
NOT NULL CHECK (VALUE >= 100000.00),
- (6) DEPT_MNG EMP_NO DEFAULT NULL
REFERENCES EMP ON DELETE SET NULL
CHECK (IF (VALUE IS NOT NULL) THEN
(SELECT COUNT(*) FROM DEPT
WHERE DEPT.DEPT_MNG = VALUE) = 1),
- (7) CHECK (DEPT_EMP_NO =
(SELECT COUNT(*) FROM EMP
WHERE DEPT_NO = EMP.DEPT_NO)),
- (8) CHECK (DEPT_TOTAL_SAL >=
(SELECT SUM(EMP_SAL) FROM EMP
WHERE DEPT_NO = EMP.DEPT_NO)));

Это определение мы обсудим в менее систематической манере, чем предыдущее. Отметим только наиболее интересные моменты. В части (3) столбец **DEPT_EMP_NO** (число служащих в отделе) определен на базовом типе **INTEGER** без значения по умолчанию, с запретом неопределенного значения и с проверочным ограничением, устанавливающим допустимый диапазон значений числа служащих в отделе. Еще одно проверочное ограничение этого столбца – (7) – вынесено на уровень определения табличного ограничения. Это ограничение устанавливает, что в каждой строке таблицы **DEPT** значение столбца **DEPT_EMP_NO** должно равняться общему числу строк таблицы **EMP**, в которых значение столбца **DEPT_NO** равно значению одноименного столбца данной строки таблицы **DEPT**.⁹ В части (5) для определения столбца **DEPT_TOTAL_SAL** (объем фонда заработной платы отдела) используется домен **SALARY**.¹⁰ Но при этом явно установлено значение столбца по умолчанию (отличное от значения по умолчанию домена), запрещено наличие неопределенных значений и введено дополнительное проверочное ограничение, определяющее нижний порог объема фонда заработной платы отдела. Еще одно проверочное ограничение – (8) – вынесено на уровень определения табличного ограничения. Это ограничение устанавливает, что в каждой строке таблицы **DEPT** значение столбца **DEPT_TOTAL_SAL** должно быть не меньше суммы значений столбца **EMP_SAL** во всех строках таблицы **EMP**, в которых значение столбца **DEPT_NO** равно значению одноименного столбца данной строки таблицы **DEPT**.¹¹ Обратите внимание на определение столбца **DEPT_MNG** – часть (6).

Этот столбец объявляется *внешним ключом* таблицы **DEPT**. Но мы хотим сказать больше. У отдела могут временно отсутствовать руководители, поэтому в столбце допускаются неопределенные значения. Но если у отдела имеется руководитель, то он должен являться руководителем только этого отдела. На первый взгляд можно было бы воспользоваться ограничением столбца **UNIQUE**. Но такое ограничение допускало бы наличие неопределенного столбца **DEPT_MNG** только в одной строке таблицы **DEPT**, а мы хотим допустить отсутствие руководителя у нескольких отделов. Поэтому потребовалось ввести более громоздкое *проверочное ограничение столбца*.

По поводу двух приведенных определений *базовых таблиц* у читателей могут возникнуть два вопроса:

- (а) почему *проверочное ограничение* (9) в первом определении и *проверочные ограничения* (7) и (8) во втором определении вынесены из определений соответствующих столбцов, хотя формально являются именно ограничениями столбцов?
- (б) почему ограничению (9) в первом определении присвоено явное имя, а ограничения (7) и (8) во втором определении оставлены безымянными?

На первый вопрос можно ответить следующим образом. Да, эти ограничения можно было бы включить в *определения столбцов*. Это дело вкуса. Но все три ограничения являются очень важными с точки зрения организации таблиц в целом. Поэтому лучше показывать их на уровне *определения табличных ограничений*.

Вот ответ на второй вопрос. Ограничение (9) в первом определении и ограничения (7) и (8) во втором определении внешне похожи, но сильно отличаются по своей сути. Ограничения (7) и (8) связаны с агрегатной семантикой столбцов **DEPT_EMP_NO** и **DEPT_TOTAL_SAL** таблицы **DEPT**. Отмена ограничений изменила бы смысл этих столбцов. Ограничение (9) является текущим административным ограничением. Если руководство предприятия примет решение разрешить использовать в одном проекте более 50 служащих, ограничение можно отменить без изменения смысла столбцов таблицы **EMP**. Имея это в виду, мы ввели явное имя ограничения (9), чтобы при необходимости имелась простая возможность отменить это ограничение с помощью оператора **ALTER TABLE**.

Наконец, определим таблицу **PRO**.

- ```
(1) CREATE TABLE PRO (
(2) PRO_NO PRO_NO PRIMARY KEY,
(3) PRO_TITLE VARCHAR(200) DEFAULT 'No title' NOT NULL,
(4) PRO_SDATE DATE DEFAULT CURRENT_DATE NOT NULL,
(5) PRO_DURAT INTERVAL YEAR DEFAULT INTERVAL '1'
YEAR NOT NULL,
(6) PRO_MNG EMP_NO UNIQUE NOT NULL
REFERENCES EMP ON DELETE NO ACTION,
(7) PRO_DESC CLOB(10M));
```

Столбец **PRO\_SDATE** содержит дату начала проекта, а столбец **PRO\_DURAT** – продолжительность проекта в годах. В этом определении имеет смысл прокомментировать часть (6). Мы считаем, что если отдел, по крайней мере временно, может существовать без руководителя, то у проекта всегда должен быть менеджер. Поэтому *определение столбца PRO\_MNG* является гораздо более строгим,

чем *определение* столбца **DEPT\_MNG** в таблице **DEPT**. Сочетание ограничений **UNIQUE** и **NOT NULL** при отсутствии значений по умолчанию приводит к абсолютной уникальности значений столбца **PRO\_MNG**. Другими словами, этот столбец обладает всеми характеристиками *первичного ключа*, хотя объявлен только как возможный ключ. Кроме того, он объявлен как внешний ключ с действием при удалении строки таблицы **EMP** с соответствующим значением *первичного ключа* **NO ACTION**, запрещающим такие удаления. В совокупности это гарантирует, что у любого проекта будет существовать менеджер, являющийся служащим предприятия. В части (7) столбец **PRO\_DESC** (описание проекта) определен как большой символьный объект с максимальным размером 10 Мбайт.

## Изменение определения базовой таблицы

Оператор изменения определения базовой таблицы **ALTER TABLE** имеет следующий синтаксис:

```
base_table_alteration ::= ALTER TABLE base_table_name
 column_alteration_action
 | base_table_constraint_alteration_action
```

Как видно из этого синтаксического правила, при выполнении одного оператора **ALTER TABLE** может быть выполнено либо действие по изменению *определения столбца*, либо действие по изменению *определения табличного ограничения* целостности.

### Добавление, изменение или удаление определения столбца

Действие по изменению *определения столбца* специфицируется в следующем синтаксисе:

```
column_alteration_action ::=
 ADD [COLUMN] column_definition
 | ALTER [COLUMN] column_name
 { SET default_definition | DROP DEFAULT }
 | DROP [COLUMN] column_name
 { RESTRICT | CASCADE }
```

Итак, с использованием оператора **ALTER TABLE** можно добавлять к определению таблицы *определение* нового столбца (действие **ADD**) и изменять или отменять *определение* существующего столбца (действия **ALTER** и **DROP** соответственно).

Смысл действия **ADD COLUMN** почти полностью совпадает со смыслом раздела *определения столбца* в операторе **CREATE TABLE**. Указывается имя нового столбца, его тип данных или домен. Могут определяться *значение по умолчанию* и *ограничения целостности*. Однако имеется одно существенное отличие: столбец, определяемый в действии **ADD** оператора **ALTER TABLE**, добавляется к уже существующей таблице, которая, скорее всего, содержит некоторый набор строк. В каждой из существующих строк новый столбец должен содержать некоторое значение, и считается, что сразу после выполнения действия **ADD** этим значением является *значение столбца по умолчанию*. Поэтому столбец, определяемый в действии **ADD**, обязательно должен иметь *значение по умолчанию*, т. е. для него недопустима ситуация, когда *значением по умолчанию* явно или неявно объявлено неопределенное значение

( **NULL** ), но среди ограничений целостности столбца присутствует ограничение **NOT NULL**.

В действии **ALTER COLUMN** можно изменить ( **SET default\_definition** ) или отменить определение значения по умолчанию для существующего столбца. Правила определения нового действующего значения столбца по умолчанию совпадают с соответствующими правилами, обсуждавшимися в подразделе *определения столбца* в операторе **CREATE TABLE**. Заметим, что изменение значения столбца по умолчанию не оказывает влияния на состояние существующих строк таблицы (даже если в некоторых из них хранится предыдущее значение столбца по умолчанию). Если столбец определен на домене, у которого существует значение по умолчанию, то после отмены определения значения столбца по умолчанию для этого столбца начинает действовать значение по умолчанию домена.

Действие **DROP COLUMN** отменяет определение существующего столбца (удаляет его из таблицы). Действие **DROP COLUMN** отвергается, если:

- (а) указанный столбец является единственным столбцом таблицы;
- (б) или в этом действии присутствует спецификация **RESTRICT**, и данный столбец используется в определении каких-либо представлений или ограничений целостности<sup>12</sup>.

Если в действии присутствует спецификация **CASCADE**, то его выполнение порождает неявное выполнение оператора **DROP** для всех представлений и ограничений целостности, в определении которых используется данный столбец.

### **Примеры изменения определения столбца**

Предположим, что на предприятии ввели систему премирования служащих. Каждый служащий может дополнительно к зарплате получать ежемесячную премию, не превышающую размер его зарплаты. Тогда разумно добавить к таблице **EMP** новый столбец **EMP\_BONUS**, используя оператор **ALTER TABLE**:

```
ALTER TABLE EMP
ADD EMP_BONUS SALARY DEFAULT NULL
CONSTRAINT BONSAL CHECK (VALUE < EMP_SAL);
```

Обратите внимание, что мы присвоили проверочному ограничению столбца явное имя, чтобы в случае, если ограничения на размер премии изменятся (что вполне возможно), можно было бы легко отменить это ограничение, воспринимая его как табличное.

При определении столбца **EMP\_SAL** таблицы **EMP** для этого столбца явно не определялось значение по умолчанию (оно наследовалось из определения домена). Если в какой-то момент это стало неправильным (например, повысился размер минимальной зарплаты), можно установить новое значение по умолчанию:

```
ALTER TABLE EMP ALTER EMP_SAL SET DEFAULT 15000.00.
```

При определении столбца **DEPT\_TOTAL\_SAL** таблицы **DEPT** для него было установлено значение по умолчанию 1000000. Главный бухгалтер предприятия может быть недоволен тем, что такие важные данные, как объем фонда зарплаты отделов, могут устанавливаться по умолчанию. Тогда можно отменить это значение по умолчанию:

**ALTER TABLE DEPT ALTER DEPT\_TOTAL\_SAL DROP DEFAULT.**

Обратите внимание, что после выполнения этого оператора при вставке новой строки в таблицу **DEPT** всегда потребуются явно указывать значение столбца **DEPT\_TOTAL\_SAL**. Хотя формально у столбца будет существовать значение по умолчанию, наследуемое от домена **SALARY** (10000.00), оно не может быть занесено в таблицу **DEPT**, поскольку противоречит ограничению столбца **DEPT\_TOTAL\_SAL CHECK (VALUE >= 100000.00)**.

Можно задуматься, действительно ли требуется поддерживать в таблице **DEPT** столбец **DEPT\_EMP\_NO**. Как мы видели, для его поддержки требуется проверять громоздкое ограничение целостности, а число служащих в любом отделе можно получить динамически с помощью простого запроса к таблице **EMP** (собственно, этот запрос входит в ограничение целостности). Поэтому может оказаться разумным отменить определение столбца **DEPT\_EMP\_NO**, выполнив следующий оператор **ALTER TABLE** :

**ALTER TABLE DEPT DROP DEPT\_EMP\_NO CASCADE.**

Напомним, что спецификация **CASCADE** ведет к тому, что при выполнении оператора будет уничтожено не только определение указанного столбца, но и определения всех ограничений целостности и представлений, в которых используется уничтожаемый столбец. В нашем случае единственное связанное с этим столбцом ограничение целостности, определенное вне определения столбца, было бы отменено, даже если бы в операторе отмены определения столбца **DEPT\_EMP\_NO** содержалась спецификация **RESTRICT**, поскольку это единственное внешнее определение ограничения является ограничением только столбца **DEPT\_EMP\_NO**.

### **Изменение набора табличных ограничений**

Действие по изменению набора табличных ограничений специфицируется в следующем синтаксисе:

```
base_table_constraint_alteration_action ::=
 ADD [CONSTRAINT] base_table_constraint_definition
 | DROP CONSTRAINT constraint_name
 { RESTRICT | CASCADE }
```

Действие **ADD [ CONSTRAINT ]** позволяет добавить к набору существующих ограничений таблицы новое ограничение целостности. Можно считать, что новое ограничение добавляется через **AND** к конъюнкции существующих ограничений, как если бы оно определялось в составе оператора **CREATE TABLE**. Но здесь имеется одно существенное отличие. Если внимательно посмотреть на все возможные виды табличных ограничений, можно убедиться, что любое из них удовлетворяется на пустой таблице. Поэтому, какой бы набор табличных ограничений ни был определен при создании таблицы, это определение является допустимым и не препятствует выполнению оператора **CREATE TABLE**. При добавлении нового табличного ограничения с использованием действия **ADD [ CONSTRAINT ]** мы имеем другую ситуацию, поскольку таблица, скорее всего, уже содержит некоторый набор строк, для которого условное выражение нового ограничения может принять значение **false**. В этом случае выполнение оператора **ALTER TABLE**, включающего действие **ADD [ CONSTRAINT ]**, отвергается.

Выполнение действия **DROP CONSTRAINT** приводит к отмене *определения* существующего табличного ограничения. Можно отменить определение только именованных табличных ограничений. Спецификации **RESTRICT** и **CASCADE** осмыслены только в том случае, если отменяемое ограничение является *ограничением возможного ключа* ( **UNIQUE** или **PRIMARY KEY** )<sup>13</sup>. При указании **RESTRICT** действие отвергается, если на данный возможный ключ ссылаются хотя бы один внешний ключ. При указании **CASCADE** действие **DROP CONSTRAINT** выполняется в любом случае, и все определения таких *внешних ключей* также отменяются.

### **Примеры изменения набора табличных ограничений**

Напомним, что мы добавили к таблице **EMP** столбец **EMP\_BONUS**, в котором сохраняются размеры ежемесячных премий служащих. Предположим, что премии выплачиваются из фонда *заработной платы* отдела, в котором работает служащий. Тогда *проверочное ограничение столбца* **DEPT\_TOTAL\_SAL**, устанавливающее, что объем фонда зарплаты отдела не должен быть меньше суммарной зарплаты служащих этого отдела, становится недостаточным, и нам требуется добавить к набору ограничений таблицы **DEPT** новое ограничение:

```
ALTER TABLE DEPT ADD CONSTRAINT TOTAL_INCOME
CHECK (DEPT_TOTAL_SAL >=
 (SELECT SUM(EMP_SAL + COALESCE(EMP_BONUS,0))
 FROM EMP WHERE EMP.DEPT_NO = DEPT_NO)).
```

Хотя это ограничение на вид довольно сложное, смысл его очень прост: суммарный доход служащих отдела не должен превышать объем зарплаты отдела. В *арифметическом выражении* под знаком агрегатной операции **SUM** используется операция **COALESCE**. Эта двуместная операция определяется следующим образом:

**COALESCE (x, y) IF x IS NOT NULL THEN x ELSE y,**

т. е. значением операции является значение первого операнда, если оно не равно **NULL**, и значение второго операнда – в противном случае. Нам пришлось воспользоваться этой операцией, поскольку в столбце **EMP\_BONUS** допускается наличие неопределенных значений.

Понятно, что новое ограничение столбца **DEPT\_TOTAL\_SAL** сильнее предыдущего, и это предыдущее ограничение можно было бы отменить. Конечно, с логической точки зрения наличие обоих ограничений ничему не повредит (предыдущее ограничение является логическим следствием нового), но при использовании не слишком интеллектуальной реализации SQL может привести к замедлению работы системы, поскольку оба ограничения могут проверяться независимо. К сожалению, при определении таблицы **EMP** мы не присвоили явное имя *проверочному ограничению столбца* **DEPT\_TOTAL\_SAL** и поэтому не можем немедленно продемонстрировать оператор отмены этого ограничения. Это не значит, что его нельзя отменить вообще. В *стандарте языка SQL* требуется, чтобы *ограничения целостности*, которым не назначены явные имена, получали имена, автоматически генерируемые системой. Любой квалифицированный пользователь SQL-ориентированной СУБД (скорее всего, администратор) может обнаружить имя любого ограничения, обратившись к системной таблице-каталогу *ограничений целостности*.

Кстати, новому ограничению мы присвоили явное имя. К этому привели следующие рассуждения. Когда создавалась исходная *схема базы данных*, руководство предприятия ничего не говорило о премиях служащих. Теперь начальство решило, что премии будут выплачиваться из фонда зарплаты. Для этого, мы добавили новый столбец и новое ограничение целостности. Но кто знает, не изменится ли снова решение о премиях? Чтобы не добавлять себе работы в будущем, дадим новому ограничению явное имя и не будем отменять предыдущее ограничение.<sup>14</sup>

При определении таблицы **EMP** было специфицировано *проверочное табличное ограничение* **PRO\_EMP\_NO**, устанавливающее, что над одним проектом не должно работать более 50 служащих. Мы уже отмечали, что это ограничение носит чисто административный характер и может быть отменено без нарушения логики базы данных. Для отмены ограничения нужно выполнить следующий оператор:

```
ALTER TABLE EMP DROP CONSTRAINT PRO_EMP_NO;
```

**Отмена определения (уничтожение) базовой таблицы**

Для отмены определения (уничтожения) *базовой таблицы* служит оператор **DROP TABLE**, задаваемый в следующем синтаксисе:

```
DROP TABLE base_table_name { RESTRICT | CASCADE }
```

Успешное выполнение оператора приводит к тому, что указанная *базовая таблица* перестает существовать. Уничтожаются все ее строки, *определения столбцов* и табличные определения целостности. При наличии спецификации **RESTRICT** выполнение оператора **DROP TABLE** отвергается, если *имя таблицы* используется в каком-либо определении представления или *ограничения целостности*<sup>15</sup>. При наличии спецификации **CASCADE** оператор выполняется в любом случае, и все определения представлений и *ограничений целостности*, содержащие ссылки на данную таблицу, также отменяются.

### Средства определения и отмены общих ограничений целостности

Виды *ограничений целостности*, с которыми мы имели дело в предыдущих разделах этой лекции, образуют иерархию ([рис. 12.2](#)).

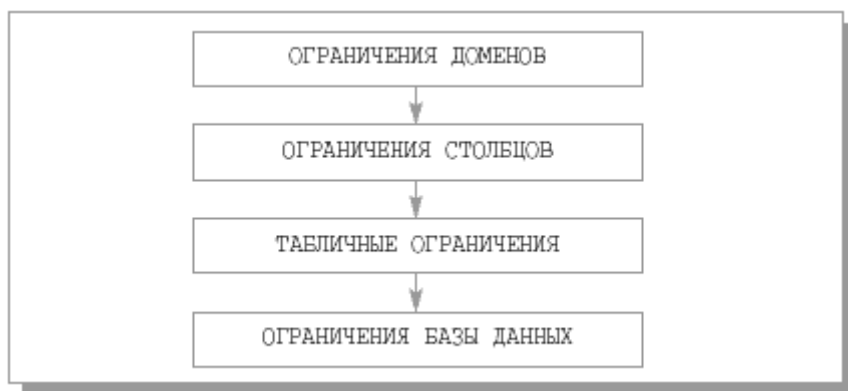


**Рис. 12.2.** Иерархия видов ограничений целостности

*Ограничения целостности*, входящие в *определение* домена, наследуются всеми столбцами, определенными на этих доменах, и являются ограничениями этих столбцов. Кроме того, в *определение столбца* могут входить определения дополнительных ограничений. *Ограничения целостности*, входящие в *определение столбца* (включая ограничения, унаследованные из определения домена), являются ограничениями таблицы, в состав определения которой входит *определение* данного столбца. Кроме того, в *определение* таблицы могут входить определения дополнительных ограничений.

Но иерархия видов ограничений *целостности* этим не исчерпывается. *Ограничения целостности*, входящие в *определение* таблицы (включая явные и унаследованные от определения доменов ограничения столбцов), представляют собой ограничения *базы данных*, частью которой является данная *таблица*. Кроме того, могут определяться дополнительные ограничения *базы данных*. В стандарте SQL такие дополнительные ограничения *базы данных* называются **ASSERTION**, а мы их будем называть *общими ограничениями целостности*.

### Определение общих ограничений целостности

Для определения *общего* ограничения *целостности* служит оператор **CREATE ASSERTION**, задаваемый в следующем синтаксисе:

```
CREATE ASSERTION constraint_name
CHECK (conditional_expression)
```

Заметим, что при создании *общего* ограничения *целостности* его имя всегда должно указываться явно. Хотя синтаксис определения общего ограничения совпадает с синтаксисом определений ограничений столбца и таблицы, в данном случае допускаются только специальные виды *условных выражений*. Мы не можем сейчас точно сформулировать свойства этих видов условий, поскольку отложили подробное рассмотрение разновидностей *условных выражений* до следующих лекций. Если говорить неформально, то особые свойства условий связаны с тем, что при определении *общих ограничений целостности* контекстом, в котором вычисляется *условное выражение*, является весь набор таблиц *базы данных*, а не набор строк таблицы, как это было при *определении табличных ограничений*. Продемонстрируем и прокомментируем несколько примеров определений *общих ограничений целостности*.

В определении таблицы **EMP** содержалось ограничение столбца **EMP\_BDATE**:

```
CHECK (EMP_BDATE >= '1917-10-24')
```

(к работе на предприятии допускаются только те лица, которые родились после Октябрьского переворота). Вот каким образом можно определить такое же ограничение на уровне *общих ограничений целостности*:

```
CREATE ASSERTION MIN_EMP_BDATE CHECK
(((SELECT MIN(EMP_BDATE)) FROM EMP) >= '1917-10-24')
```

В логическом условии этого общего ограничения выбирается минимальное значение столбца **EMP\_BDATE** (дата рождения самого старого служащего). Значением *условного выражения* будет **false** в том и только в том случае, если среди служащих имеется хотя бы один, родившийся до указанной даты.

Теперь переформулируем в виде *общего ограничения целостности* ограничение таблицы **EMP** **PRO\_EMP\_NO**, которое определялось следующим образом:

```
CONSTRAINT PRO_EMP_NO CHECK
((SELECT COUNT (*) FROM EMP E
 WHERE E.PRO_NO = PRO_NO) <= 50)
```

(над одним проектом не может работать более 50 служащих).

Вот формулировка эквивалентного *общего ограничения целостности*:

```
CREATE ASSERTION NEW_PRO_EMP_NO CHECK
(NOT EXISTS (SELECT PRO_NO FROM EMP GROUP BY PRO_NO
 HAVING COUNT(*) > 50)).
```

Логическое выражение этого ограничения может принимать только значения **true** и **false**. Внутренний оператор выборки группирует строки таблицы **EMP** таким образом, что в одну группу попадают все строки с одинаковым значением столбца **PRO\_NO**. Затем эти группы фильтруются по условию раздела **HAVING**, и остаются только группы, включающие более 50 строк. В результирующей таблице содержатся строки из одного столбца, содержащего значение **PRO\_NO** оставшихся групп. Предикат **NOT EXISTS** принимает значение **true** тогда и только тогда, когда эта результирующая таблица не содержит ни одной строки, т. е. нет ни одного проекта, в котором работает больше 50 служащих.

Покажем, как можно сформулировать в виде *общего ограничения целостности* ограничение внешнего ключа. Например, приведем такую эквивалентную формулировку для определения внешнего ключа **PRO\_NO**, входящего в состав определения таблицы **EMP**:

```
FOREIGN KEY PRO_NO REFERENCES PRO (PRO_NO)
```

В виде *общего ограничения целостности* это может выглядеть следующим образом:

```
(1) CREATE ASSERTION FK_PRO_NO CHECK
(2) (NOT EXISTS (SELECT * FROM EMP
 WHERE PRO_NO IS NOT NULL AND
(3) NOT EXISTS (SELECT * FROM PRO
(4) WHERE PRO.PRO_NO = EMP.PRO_NO))).
```

Логическое выражение этого ограничения выглядит достаточно сложным и нуждается в пояснении. Условие выборки оператора **SELECT** на строке (2) состоит из двух частей, связанных через **AND**. Первая часть отфильтровывает те строки таблицы **EMP**, у которых в столбце **PRO\_NO** содержится **NULL**. Если этот столбец содержит **NULL** во всех строках таблицы, то результирующая таблица оператора выборки на строке (2) будет пустой, и значением предиката **NOT EXISTS** будет **true**, т. е. ограничение удовлетворяется.

Теперь предположим, что в таблице **EMP** нашлась строка **emp**, в столбце **PRO\_NO** которой содержится значение, отличное от **NULL**. Назовем это значение **cand\_pro\_no**. Для него вычисляется вторая часть условия выборки

оператора **SELECT** на строке (2). Оператор выборки на строке (3) выбирает все строки таблицы **PRO**, значение столбца **PRO\_NO** которых равняется **cand\_pro\_no**. Если для данного значения **cand\_pro\_no** нашлась хотя бы одна такая строка, то результирующая таблица оператора выборки на строке (3) будет непустой, и значением предиката **NOT EXISTS** на строке (3) будет **false**. Соответственно, все условие выборки первого оператора **SELECT** примет значение **false**, и строка со значением **cand\_pro\_no** в столбце **PRO\_NO** будет отфильтрована.<sup>16</sup>

Если же найдется хотя бы одна строка таблицы **EMP** с таким значением **cand\_pro\_no** столбца **PRO\_NO**, что в таблице **PRO** не найдется ни одной строки, значение столбца **PRO\_NO** которой равнялось бы этому **cand\_pro\_no**, то результирующая таблица оператора выборки на строке (3) будет пустой, и значением предиката **NOT EXISTS** на строке (3) будет **true**. Тогда все условие выборки первого оператора **SELECT** примет значение **true**, и эта строка таблицы **EMP** будет пропущена в результирующую таблицу. Значением предиката **NOT EXISTS** будет **false**, т. е. ограничение не удовлетворяется.

Мы сознательно привели такое подробное пояснение не только для того, чтобы прояснить смысл *условного выражения общего ограничения целостности* **FK\_PRO\_NO**, но и чтобы дать понять, во что реально вырождается простая синтаксическая конструкция определения *внешнего ключа*. Как показывает опыт, многие начинающие проектировщики SQL-ориентированных баз данных думают, что ссылочные ограничения так же легко поддерживать, как определять.

Наконец, сформулируем общее ограничение целостности, состоящее в том, что никакой *менеджер проекта* не должен иметь суммарный общий доход, больший суммарного дохода руководителя отдела, в котором работает этот менеджер.

```
(1) CREATE ASSERTION PRO_MNG_CONSTR CHECK
(2) NOT EXISTS (SELECT * FROM EMP EMP1, EMP EMP2,
(3) DEPT, PRO WHERE
(4) EMP1.EMP_NO = PRO.PRO_MNG AND
(5) EMP1.DEPT_NO = DEPT.DEPT_NO AND
(6) DEPT.DEPT_MNG = EMP2.EMP_NO AND
(7) EMP1.EMP_SAL + COALESCE (EMP1.EMP_BONUS,0) >
(8) EMP2.EMP_SAL + COALESCE (EMP2.EMP_BONUS,0);
```

В логическом выражении этого ограничения используется оператор выборки **SELECT**, в разделе перечня таблиц (**FROM**) впервые в этом курсе используется несколько таблиц. Такие запросы в SQL называются запросами с соединениями, и мы воспользуемся случаем, чтобы пояснить на примере (конечно, предварительно), как их следует понимать в соответствии со стандартом языка SQL.

Итак, в разделе **FROM** оператора выборки, используемого в логическом условии этого ограничения, через запятую перечислены четыре элемента – **EMP EMP1**, **EMP EMP2**, **DEPT** и **PRO**. Выражение вида **EMPANOTHER\_NAME** означает применение своего рода операции переименования. Внутри запроса столбцы этого "экземпляра" **EMP** имеют "квалифицированные" имена вида **ANOTHER\_NAME.column\_name**, где **column\_name** обозначает имя существующего столбца таблицы **EMP**.

Вычисление оператора выборки начинается с того, что формируется расширенное *декартово произведение* всех таблиц, указанных в разделе **FROM**. В данном случае схема результирующей *таблицы раздела FROM* будет содержать следующие имена столбцов: EMP1.EMP\_NO, EMP1.EMP\_NAME, EMP1.EMP\_BDATE, EMP1.EMP\_SAL, EMP1.EMP\_BONUS, EMP1.DEPT\_NO, EMP1.PRO\_NO, EMP2.EMP\_NO, EMP2.EMP\_NAME, EMP2.EMP\_BDATE, EMP2.EMP\_SAL, EMP2.EMP\_BONUS, EMP2.DEPT\_NO, EMP2.PRO\_NO, DEPT.DEPT\_NO, DEPT.DEPT\_EMP\_NO, DEPT.DEPT\_TOTAL\_SAL, DEPT.DEPT\_MNG, PRO.PRO\_NO, PRO.PRO\_TITLE, PRO.PRO\_SDATE, PRO.PRO\_DURAT, PRO.PRO\_O\_MNG, PRO\_DESC. Для удобства назовем эту "широкую" таблицу **ALL\_TOGETHER**.<sup>17</sup>

Условие раздела **WHERE** состоит из четырех частей, связанных через **AND**. Обсудим их последовательно. После проверки условия **EMP1.EMP\_NO = PRO.PRO\_MNG** в таблице **ALL\_TOGETHER** останутся все служащие-менеджеры проектов вместе со своими проектами в комбинации со всеми возможными отделами и всеми возможными служащими (назовем эту отфильтрованную таблицу **ALL\_TOGETHER\_STEP1**). После проверки условия **EMP1.DEPT\_NO = DEPT.DEPT\_NO** в таблице **ALL\_TOGETHER\_STEP1** останутся все служащие-менеджеры проектов вместе со своими проектами и вместе с описанием своих отделов в комбинации со всеми возможными служащими (назовем эту отфильтрованную таблицу **ALL\_TOGETHER\_STEP2**). После проверки условия **DEPT.DEPT\_MNG = EMP2.EMP\_NO** в таблице **ALL\_TOGETHER\_STEP2** останутся все служащие-менеджеры проектов вместе со своими проектами, вместе с описанием своих отделов и вместе с руководителями этих отделов (по одной строке для каждого допустимого сочетания " *проект-менеджер\_проекта-отдел\_менеджера\_проекта-руководитель\_отдела\_менеджера\_проекта* "). Назовем эту отфильтрованную таблицу **ALL\_TOGETHER\_STEP3**. Легко видеть, что после проверки условия **EMP1.EMP\_SAL + EMP1.EMP\_BONUS > EMP2.EMP\_SAL + EMP2.EMP\_BONUS** в таблице **ALL\_TOGETHER\_STEP3** могут остаться только строки *проект-менеджер\_проекта-отдел\_менеджера\_проекта-руководитель\_отдела\_менеджера\_проекта*, в которых суммарный доход менеджера проекта превышает суммарный доход руководителя отдела, где работает *менеджер проекта*. Если хотя бы одна такая строка существует, то результат оператора выборки будет непустым, значением предиката **NOT EXISTS** будет **false**, и тем самым ограничение целостности **PRO\_MNG\_CONSTR** будет нарушено.

### Отмена определения общего ограничения целостности

Для того чтобы отменить ранее определенное общее ограничение целостности, нужно воспользоваться оператором **DROP ASSERTION**, задаваемым в следующем синтаксисе:

**DROP ASSERTION constraint\_name**

Вот пример оператора, отменяющего определение дискриминационного *общего ограничения целостности* **PRO\_MNG\_CONSTR**:

**DROP ASSERTION PRO\_MNG\_CONSTR;**

### Немедленная и откладываемая проверка ограничений

На первый взгляд кажется, что *ограничения целостности* (всех видов) должны немедленно проверяться в случае выполнения любого действия, изменяющего содержимое базы данных (вставка в любую таблицу новой строки, изменение или

удаление существующих строк). Однако можно определить такие *ограничения целостности*, логическое выражение которых будет принимать значение **false** при любой *немедленной проверке*. Одним из примеров такого ограничения является ограничение

```
CHECK (DEPT_EMP_NO =
 (SELECT COUNT(*) FROM EMP
 WHERE DEPT_NO = EMP.DEPT_NO))
```

из определения таблицы **DEPT**. Предположим, например, что в отдел зачисляется новый служащий. Тогда нужно выполнить две операции: (а) вставить новую строку в таблицу **EMP** и (б) изменить соответствующую строку таблицы **DEPT** (прибавить единицу к значению столбца **DEPT\_EMP\_NO**). Очевидно, что в каком бы порядке ни выполнялись эти операции, сразу после выполнения первой из них ограничение целостности будет нарушено, соответствующее действие будет отвергнуто, и мы никогда не сможем принять на работу нового служащего.

Поскольку *ограничения целостности, немедленная проверка* которых бессмысленна, являются нужными и полезными, в язык SQL включены средства, позволяющие регулировать время проверки ограничений. Если говорить более точно, в контексте каждой выполняемой транзакции каждое ограничение целостности должно находиться в одном из двух режимов: режиме *немедленной проверки* (**immediate**) или режиме *отложенной проверки* (**deferred**). Все *ограничения целостности*, находящиеся в режиме *немедленной проверки*, проверяются при выполнении в транзакции любой операции, изменяющей состояние базы данных. Если действие операции нарушает какое-либо немедленно проверяемое ограничение целостности, то это действие отвергается.<sup>18</sup> *Ограничения целостности*, находящиеся в режиме *отложенной проверки*, проверяются при завершении транзакции (выполнении операции **COMMIT**)<sup>19</sup>. Если действия этой транзакции нарушают какое-либо отложено проверяемое ограничение целостности, то транзакция откатывается (операция **COMMIT** трактуется как операция **ROLLBACK**).<sup>20</sup>

Для этого в качестве заключительной синтаксической конструкции к любому определению *ограничения целостности* (любого вида) может быть добавлена спецификация **INITIALLY** в следующей синтаксической форме:

```
INITIALLY { DEFERRED | IMMEDIATE }
[[NOT] DEFERRABLE]
```

Эта спецификация указывает, в каком режиме должно находиться данное ограничение целостности в начале выполнения любой транзакции (**INITIALLY IMMEDIATE** означает, что в начале выполнения транзакции данное ограничение будет находиться в режиме *немедленной проверки*, а **INITIALLY DEFERRED** – что в начале любой транзакции ограничение будет находиться в режиме *отложенной проверки*), а также возможности смены режима этого ограничения при выполнении транзакции (**DEFERRABLE** означает, что для данного ограничения может быть установлен режим *отложенной проверки*, а **NOT DEFERRABLE** – что не может<sup>21</sup>).

Комбинация **INITIALLY DEFERRED NOT DEFERRABLE** является недопустимой. Если в определении ограничения спецификация начального режима проверки отсутствует, то подразумевается наличие спецификации **INITIALLY IMMEDIATE**. При наличии явной

или неявной спецификации **INITIALLY IMMEDIATE** и отсутствии явного указания возможности смены режима подразумевается наличие спецификации **NOT DEFERRABLE**. При наличии спецификации **INITIALLY DEFERRED** и отсутствии явного указания возможности смены режима подразумевается наличие спецификации **DEFERRABLE**.

При выполнении транзакции можно изменить режим проверки некоторых или всех *ограничений целостности* для данной транзакции. Для этого используется оператор **SET CONSTRAINTS**, задаваемый в следующем синтаксисе:

```
SET CONSTRAINTS { constraint_name_commalist | ALL }
 { DEFERRED | IMMEDIATE }
```

Если в операторе указывается список имен *ограничений целостности*, то все они должны быть **DEFERRABLE**; если хотя бы для одного ограничения из списка это требование не выполняется, то операция **SET CONSTRAINTS** отвергается. При указании ключевого слова **ALL** режим устанавливается для всех ограничений, в определении которых явно или неявно было указано **DEFERRABLE**. Если в качестве желаемого режима проверки ограничений задано **DEFERRED**, то все указанные ограничения переводятся в режим отложенной проверки. Если в качестве желаемого режима проверки ограничений задано **IMMEDIATE**, то все указанные ограничения переводятся в режим *немедленной проверки*. При этом если хотя бы одно из этих ограничений не удовлетворяется, то операция **SET CONSTRAINTS** отвергается, и все указанные ограничения остаются в предыдущем режиме.

При выполнении операции **COMMIT** неявно выполняется операция **SET CONSTRAINTS ALL IMMEDIATE**. Если эта операция отвергается, то **COMMIT** срабатывает как **ROLLBACK**.

## Заключение

В этой и предыдущей лекциях мы обсудили наиболее важные аспекты языка **SQL**, связанные с определением схемы *базы данных*, – типы данных **SQL**, средства определения доменов, *базовых таблиц* и *ограничений целостности*. Кроме того, были рассмотрены средства **SQL**, позволяющие динамически изменять и удалять определения этих объектов. Язык **SQL** устроен таким образом, что практически невозможно изложить какую-либо его часть независимо от других частей. И хотя эти две лекции по смыслу должны быть первыми среди лекций, посвященных **SQL** (было бы странно обсуждать *операторы* выборки строк из таблиц, вставки, изменения и удаления строк до обсуждения средств создания таблиц и *ограничений целостности*), нам пришлось забежать вперед и воспользоваться материалом следующих лекций для объяснения средств определения *ограничений целостности*. Надеюсь, что это не создало слишком больших неудобств для читателей, и отсутствие формальных определений удалось компенсировать наличием простых примеров.

## Лабораторная работа3. Общая характеристика оператора SELECT и организация списка ссылок на таблицы в разделе FROM

### Введение

Несмотря на то, что язык *SQL* является полным языком баз данных, включающим множество разнообразных средств определения схемы, ограничения и поддержки целостности *базы данных*, поддержки администрирования, заполнения и модификации таблиц *базы данных*, поддержки разработки приложений и т. д., для подавляющего большинства пользователей этот язык остается *языком запросов*, т. е. языком, позволяющим формулировать произвольно сложные и точные декларативные запросы к базе данных.

Как отмечалось в конце предыдущей лекции, структура *стандарта языка SQL* фактически не позволяет описать одну часть языка (в частности, средства запросов) в отрыве от других частей. Тем не менее, полагая, что средства *выборки данных* составляют наиболее интересную и практически значимую часть языка, мы выделили для их рассмотрения несколько отдельных лекций.

Напомним, что в этом курсе мы ограничиваемся базовым подмножеством *SQL:1999* и *SQL:2003* ("прямым *SQL*") и даже это *подмножество* описываем не в полном объеме стандарта. Кроме того, в данной лекции мы не будем точно придерживаться порядка введения понятий и синтаксических конструкций, принятого в *стандарте языка*. Мы начнем с некоторой общей картины, дающей *представление* об операторе выборки, а затем будем постепенно уточнять ее.

## Скалярные выражения

**Скалярное выражение**<sup>1</sup> - это выражение, вырабатывающее результат некоторого типа, специфицированного в стандарте. *Скалярные выражения* являются основой языка *SQL*, поскольку, хотя это реляционный язык, все условия, элементы списков выборки и т. д. базируются именно на *скалярных выражениях*. В *SQL:1999* имеется несколько разновидностей *скалярных выражений*. К числу наиболее важных разновидностей относятся *численные выражения*; выражения со значениями-строками символов; выражения со значениями даты-времени; выражения со значениями-временными интервалами; *булевские выражения*. Мы не будем слишком глубоко вникать в тонкости, но тем не менее приведем некоторые базовые спецификации и пояснения.

Прежде чем перейти к конкретным видам *скалярных выражений*, рассмотрим некоторые наиболее общие языковые конструкции, на которых эти выражения базируются.

## Общие синтаксические правила построения скалярных выражений

В *SQL:2003* имеются девять разновидностей выражений в соответствии с девятью категориями типов данных, значения которых вырабатываются при *вычислении выражения*

```
value_expression ::=
 numeric_value_expression
 | string_value_expression
 | datetime_value_expression
 | interval_value_expression
 | boolean_value_expression
 | array_value_expression
 | multiset_value_expression
 | row_value_expression
```

| user\_defined\_value\_expression  
| reference\_value\_expression

Как уже отмечалось в начале этого раздела, мы ограничимся обсуждением первых пяти разновидностей выражений. В основе построения этих видов выражений лежит *первичное выражение*, определяемое следующим синтаксическим правилом:

value\_expression\_primary ::=  
    unsigned\_value\_specification  
    | column\_reference  
    | set\_function\_specification  
    | scalar\_subquery  
    | case\_expression  
    | (value\_expression)  
    | cast\_specification

В пределах этого курса можно считать, что спецификация беззнакового значения (*unsigned\_value\_specification*) - это всегда *литерал* соответствующего типа или вызов *ниладической* функции (например, *CURRENT\_USER*)<sup>2</sup>. При *вычислении выражения V* для строки таблицы каждая ссылка на столбец (*column\_reference*) этой таблицы, непосредственно содержащаяся в *V*, рассматривается как ссылка на значение данного столбца в данной строке. *Агрегатные функции (функции над множествами - set\_function\_specification)* обсуждаются в следующих лекциях. Если *первичное выражение* является *скалярным подзапросом (scalar\_subquery, или подзапросом, результатом которого является таблица, состоящая из одной строки и одного столбца)* и результат подзапроса пуст, то результат *первичного выражения - неопределенное значение*. (*Подзапросы* обсуждаются в следующей лекции, *выражения с переключателем (case\_expression)* рассматриваются ниже в этом разделе.)

## Численные выражения

**Численное выражение** - это выражение, значение которого относится к числовому типу данных. Вот формальный синтаксис *численного выражения*:

numeric\_value\_expression> ::= numeric\_term  
    | numeric\_value\_expression + term  
    | numeric\_value\_expression - term  
numeric\_term ::= numeric\_factor  
    | numeric\_term \* numeric\_factor  
    | numeric\_term / numeric\_factor  
numeric\_factor ::= [ { + | - } ] numeric\_primary  
numeric\_primary ::= value\_expression\_primary  
    | numeric\_value\_function

Следует обратить внимание на то, что в *численных выражениях SQL* первичная составляющая (*numeric\_primary*) является либо *первичным выражением* (см. выше), либо *вызовом функции с численным значением (numeric\_value\_function)*. Из этого, в частности, следует, что в *численные выражения* могут входить *выражения с переключателем* и операции *преобразования типов*. *Вызовы функций с численным значением* определяются следующими синтаксическими правилами:

```

numeric_value_function ::=
 POSITION (character_value_expression
 IN character_value_expression)
 | {CHAR_LENGTH | CHARACTER_LENGTH }
 (string_value_expression)
 | OCTET_LENGTH (string_value_expression)
 | BIT_LENGTH (string_value_expression)
 | EXTRACT ({ datetime_field | time_zone field }
 FROM { datetime_value_expression
 | interval_value_expression })
 | CARDINALITY (array_value_expression
 | multiset_value_expression)
 | ABS (numeric_value_expression)
 | MOD (numeric_value_expression)

```

Мы достаточно подробно обсуждали функции определения позиции и длины по отношению к символьным и битовым строкам при рассмотрении соответствующих типов данных; здесь приводится только уточненный синтаксис их вызова. Функция *EXTRACT* извлечения поля из значений дата-время или интервал позволяет получить в виде точного числа с масштабом 0 значение любого поля (года, месяца, дня и т. д.). Какой конкретный тип точных чисел будет выбран - определяется в реализации. Функции *ABS* и *MOD* возвращают абсолютное значение числа и остаток от деления одного целого значения на другое соответственно.

### **Выражения, значениями которых являются символьные или битовые строки**

*Выражения символьных и битовых строк* - это выражения, значениями которых являются символьные или битовые строки. Соответствующие конструкции определяются следующим синтаксисом:

```

string_value_expression ::= character_value_expression
 | bit_value_expression
character_value_expression ::= concatenation
 | character_factor
concatenation ::= character_value_expression || character_factor
character_factor ::= character_primary [collate_clause]
character_primary ::= value_expression_primary
 | string_value_function
bit_value_expression ::= bit_concatenation
 | bit_factor
bit_concatenation ::= bit_value_expression || bit_primary
bit_primary ::= value_expression_primary
 | string value function

```

Если не вдаваться в тонкости, смысл *выражений символьных и битовых строк* понятен из описания синтаксиса: единственная применимая для построения выражений операция - это конкатенация, производящая "склейку" строк-операндов. Более важно то, что первичной составляющей выражения над строками может быть как первичное *скалярное*

*выражение* (см. выше), так и *вызов функций, возвращающих строчные значения*. Репертуар и синтаксис вызова таких функций определяются следующими правилами:

```
string_value_function ::= character_value_function
 | bit_value_function
character_value_function ::= SUBSTRING
 (character_value_expression
 FROM start_position
 [FOR string_length])
 | SUBSTRING (character_value_expression
 SIMILAR character_value_expression
 ESCAPE character_value_expression)
 | { UPPER | LOWER }
 (character_value_expression)
 | CONVERT (character_value_expression
 USING conversion_name)
 | TRANSLATE (character_value_expression
 USING translation_name)
 | TRIM ([{ LEADING | TRAILING | BOTH }]
 [character_value_expression]
 [character_value_expression])
 | OVERLAY (character_value_expression
 PLACING character_value_expression
 FROM start_position
 [FOR string_length])
bit_value_function ::= SUBSTRING (bit_value_expression
 FROM start_position
 [FOR string_length])
start_position ::= numeric_value_expression
string_length ::= numeric_value_expression
```

Основные полезные функции - *выделение подстроки* ( ***SUBSTRING*** ) и замена малых букв на заглавные и наоборот ( ***UPPER*** и ***LOWER*** ) - мы упоминали при рассмотрении типов символьных и битовых строк. Обсуждение функции ***SUBSTRING ... SIMILAR ... ESCAPE*** отложим до следующей лекции. Как видно из описания синтаксиса функций, возвращающих строчные значения, для *символьных строк* имеются еще четыре функции: ***CONVERT***, ***TRANSLATE***, ***TRIM*** и ***OVERLAY***. По смыслу все они очень просты. Функция ***CONVERT*** меняет *кодировку символов* в заданной строке, причем набор символов не меняется. Способ задания правил перекодировки определяется в реализации. Функция ***TRANSLATE***, наоборот, в соответствии с правилами *трансляции* "переводит" текстовую строку на другой язык (используя набор символов целевого алфавита). Кодировка не меняется. Функция ***TRIM*** "отсекает" последовательности указанного символа в начале, в конце или в начале и в конце заданной строки. Наконец, функция ***OVERLAY*** заменяет указанную подстроку первого операнда строкой, заданной в качестве второго операнда.

## Выражения даты-времени

К **выражениям даты-времени** мы относим выражения, вырабатывающие значения *типа* дата-время и интервал. *Выражения даты-времени* определяются следующими синтаксическими правилами:

```
datetime_value_expression ::=
 datetime_term
 | interval_value_expression + datetime term
 | datetime_value_expression + interval term
 | datetime value expression - interval term
datetime_term ::= datetime_primary
 [AT { LOCAL | TIME_ZONE interval_value_expression }]
datetime_primary ::= value_expression_primary
 | datetime_value_function
```

Как видно из описания синтаксиса, сами выражения строятся очень просто - на основе обычных *арифметических операций*. Снова более интересны первичные составляющие - *вызовы функций, возвращающих значение дата-время*. Эти вызовы определяются следующим синтаксисом:

```
datetime_value_function ::= CURRENT_DATE
 | CURRENT_TIME [(precision)]
 | LOCALTIME [(precision)]
 | CURRENT_TIMESTAMP [(precision)]
 | LOCALTIMESTAMP [(precision)]
```

Видимо, приведенные синтаксические правила не нуждаются в комментариях: можно получить текущую дату, а также текущее время с желаемой точностью. Отличие функций **LOCALTIME** и **LOCALTIMESTAMP** от **CURRENT\_TIME** и **CURRENT\_TIMESTAMP**, соответственно, состоит в том, что первая *пара функций* не возвращает смещение локального времени от Гринвича.

Синтаксис выражений со значениями типа интервал определяется следующими правилами:

```
interval_value_expression ::=
 interval_term
 | interval_value_expression + interval term
 | interval_value_expression - interval term
 | (datetime value expression - datetime term)
 interval_qualifier
interval_term ::= interval_factor
 | interval_term * numeric_factor
 | interval_term / numeric_factor
 | numeric_term * interval_factor
interval_factor ::= [{ + | - }]
 interval_primary [<interval qualifier>]
interval_primary ::= value_expression_primary
 | interval_value_function
```

Как видно из приведенных правил, выражения со значениями типа интервал устроены очень просто; почти вся содержательная информация была приведена при обсуждении соответствующего типа данных. Стоит только заметить, что квалификатор интервала указывается для того, чтобы явно специфицировать *единицу измерения* интервала. Поддерживается только одна функция **ABS** (абсолютное значение), аргументом которой является выражение со значением типа интервал.

## Булевские выражения

К **булевским выражениям** относятся выражения, вырабатывающие значения *булевого типа* (напомним, что *булевский тип* языка SQL содержит три логических значения - **true**, **false** и **unknown**). Булевские выражения определяются следующими синтаксическими правилами:

```
boolean_value_expression ::= boolean_term
 | boolean_value_expression OR boolean_term
boolean_term ::= boolean_factor
 | boolean_term AND boolean_factor
boolean_factor ::= [NOT] boolean_test
boolean_test ::= boolean_primary [IS [NOT] truth_value]
truth_value ::= TRUE | FALSE | UNKNOWN
boolean_primary ::= predicate
 | (boolean_value_expression)
 | value_expression_primary
```

Выражения вычисляются слева направо с учетом *приоритетов операций* (наиболее высокий приоритет имеет *унарная операция NOT*, следующим уровнем приоритета обладает "мультипликативная" операция конъюнкции **AND**, и самый низкий приоритет у "аддитивной" операции дизъюнкции **OR**) и круглых скобок. Операции **IS** и **IS NOT** определяются следующими *таблицами истинности*:

| IS      | TRUE  | FALSE | UNKNOWN |
|---------|-------|-------|---------|
| TRUE    | TRUE  | FALSE | FALSE   |
| FALSE   | FALSE | TRUE  | FALSE   |
| UNKNOWN | FALSE | FALSE | TRUE    |
| IS NOT  | TRUE  | FALSE | UNKNOWN |
| TRUE    | FALSE | TRUE  | TRUE    |

|                |             |              |              |
|----------------|-------------|--------------|--------------|
| <b>FALSE</b>   | <b>TRUE</b> | <b>FALSE</b> | <b>TRUE</b>  |
| <b>UNKNOWN</b> | <b>TRUE</b> | <b>TRUE</b>  | <b>FALSE</b> |

### Выражения с переключателем

*Выражения с переключателем* в некотором смысле ортогональны рассмотренным выше видам выражений, поскольку разные *выражения с переключателем* могут вырабатывать значения разных типов в зависимости от типа данных элементов. Поскольку мы еще вообще не рассматривали этот вид выражений, обсудим их более подробно. Как обычно, начнем с синтаксиса:

```
case_expression ::= case_abbreviation
 | case_specification
```

```
case_abbreviation ::= NULLIF (value_expression , value_expression)
 | COALESCE (value_expression_comma_list)
```

```
case specification ::= simple_case | searched_case
```

```
simple_case ::= CASE value_expression simple_when_clause_list
 [ELSE value_expression] END
```

```
searched_case ::= CASE searched_when_clause_list
 [ELSE value_expression] END
```

```
simple_when_clause ::= WHEN value_expression
 THEN value_expression
```

```
searched_when_clause ::= WHEN conditional_expression
 THEN value_expression
```

Наиболее общим видом *выражения с переключателем* является *выражение с поисковым переключателем* (**searched\_case**). Правила вычисления *выражений* этого вида состоят в следующем. Вычисляется *логическое выражение*, указанное в первом разделе **WHEN** списка (**searched\_when\_clause\_list**). Если значение этого *логического выражения* равняется **true**, то значением всего *выражения с поисковым переключателем* является значение *выражения*, указанного в первом разделе **WHEN** после ключевого слова **THEN**. Иначе аналогичные действия производятся для второго раздела **WHEN** и т. д. Если ни для одного раздела **WHEN** при вычислении *логического выражения* не было получено значение **true**, то значением всего *выражения с поисковым переключателем* является значение *выражения*, указанного в разделе **ELSE**. Типы всех выражений, значения которых могут являться результатом *выражения с поисковым переключателем*, должны быть совместимыми, и типом результата является "наименьший общий" тип набора типов выражений-кандидатов на выработку результата<sup>3</sup>. Если в выражении отсутствует раздел **ELSE**, предполагается наличие раздела **ELSE NULL**.

В *выражении с простым переключателем* (**simple\_case**) тип данных операнда переключателя (*выражения, непосредственно следующего за ключевым словом CASE*, назовем его **CO - Case Operand**) должен быть совместим с типом данных операнда

каждого варианта (выражения, непосредственно следующего за ключевым словом **WHEN** ; назовем **WO - When Operand** ). *Выражение с простым переключателем*

```
CASE CO WHEN WO1 THEN result1
 WHEN WO2 THEN result2

 WHEN WOn THEN resultn
 ELSE result
END
```

*эквивалентно выражению с поисковым переключателем*

```
CASE WHEN CO = WO1 THEN result1
 WHEN CO = WO2 THEN result2

 WHEN CO = WOn THEN resultn
 ELSE result
END
```

Выражение **NULLIF (V1, V2)** эквивалентно следующему *выражению с переключателем*:

```
CASE WHEN V1 = V2 THEN NULL ELSE V1 END.
```

Выражение **COALESCE (V1, V2)** эквивалентно следующему *выражению с переключателем*:

```
CASE WHEN V1 IS NOT NULL THEN V1 ELSE V2 END.
```

Выражение **COALESCE (V1, V2, . . . Vn)** для  $n \geq 3$  эквивалентно следующему *выражению с переключателем*:

```
CASE WHEN V1 IS NOT NULL THEN V1 ELSE COALESCE (V2,... n) END.
```

### **Общая структура оператора выборки в языке SQL**

Для *выборки данных* в *прямом SQL* используется *оператор SELECT*, возвращающий набор<sup>4</sup> из одной или нескольких строк одинаковой структуры и задаваемый в следующем синтаксисе:

```
SELECT [ALL | DISTINCT] select_item_commalist
 FROM table_reference_commalist
 [WHERE conditional_expression]
 [GROUP BY column_name_commalist]
 [HAVING conditional_expression]
 [ORDER BY order_item_commalist]
```

### **Семантика оператора выборки**

Для начала опишем общую схему выполнения *оператора SELECT* в соответствии с предписаниями стандарта<sup>5</sup> Выполнение запроса состоит из нескольких шагов, соответствующих разделам оператора выборки. На первом шаге выполняется *раздел*

*FROM*. Если список ссылок на таблицы (*table\_reference\_commalist*) этого раздела соответствует таблицам *A, B, ... C*<sup>6</sup>, то в результате выполнения *раздела FROM* образуется таблица (назовем ее *T*), являющаяся расширенным декартовым произведением таблиц *A, B, ..., C*. Если в *разделе FROM* указана только одна таблица, то она же и является результатом выполнения этого раздела. Как говорилось в лекции 3, в *реляционной алгебре* для корректного выполнения операции взятия расширенного *декартова произведения* отношений в общем случае требуется применение *операции переименования атрибутов*. Соответствующие возможности переименования столбцов таблиц, указанных в списке *раздела FROM*, поддерживаются и в SQL. Альтернативный способ именования столбцов результирующей таблицы *T* основывается на использовании квалифицированных имен столбцов. Идея этого подхода (более раннего в истории SQL) заключается в том, что с любой таблицей, ссылка на которую содержится в списке *раздела FROM*, можно связать некоторое имя-*псевдоним* (в стандарте оно называется *correlation name*<sup>7</sup>). Тогда, если с такой таблицей *A* связан *псевдоним Z*, то в пределах оператора выборки можно ссылаться на любой столбец *a* таблицы *A* по квалифицированному имени *Z.a*. Мы обсудим это подробнее в следующем подразделе. Пока же будем считать, что имена всех столбцов таблицы *T* определены и различны.

На втором шаге выполняется *раздел WHERE*. *Условное выражение* (*conditional\_expression*) этого раздела применяется к каждой строке таблицы *T*, и результатом является таблица *T1*, содержащая те и только те строки таблицы *T*, для которых результатом вычисления *условного выражения* является *true*. (Заголовки таблиц *T* и *T1* совпадают.) Если *раздел WHERE* в операторе выборки отсутствует, то это трактуется как наличие *раздела WHERE true*<sup>8</sup> т. е. *T1* содержит те и только те строки, которые содержатся в таблице *T*. Обратите внимание на разницу в трактовке *логических выражений* в операторах выборки и в табличных *ограничениях целостности*. *Логическое выражение* *раздела WHERE* (и *раздела HAVING*) оператора выборки разрешает выборку строки в том и только в том случае, когда результатом вычисления *логического выражения* на данной строке является *true* (значения *false* и *unknown* не являются разрешающими). *Логическое выражение* табличного ограничения целостности запрещает наличие строки в таблице в том и только в том случае, когда результатом вычисления *логического выражения* на данной строке является *false* (значения *true* и *unknown* не являются запрещающими).

Если в операторе выборки присутствует *раздел GROUP BY*, то он выполняется на третьем шаге. Каждый элемент списка имен столбцов (*column\_name\_commalist*), указываемого в этом разделе, должен быть одним из имен столбцов таблицы *T1*. В результате выполнения *раздела GROUP BY* образуется *сгруппированная таблица T2*, в которой строки таблицы *T1* расставлены в минимальное число групп, таких, что во всех строках одной группы значения столбцов, указанных в списке имен столбцов *раздела GROUP BY* (*столбцов группировки*), одинаковы<sup>9</sup>. Заметим, что сгруппированные таблицы не могут являться окончательным результатом оператора выборки. Они существуют только на концептуальном уровне на стадии выполнения запроса, содержащего *раздел GROUP BY*.

Если в операторе выборки присутствует *раздел HAVING*, то он выполняется на следующем шаге. *Условное выражение* этого раздела применяется к каждой группе строк таблицы *T2*, и результатом является сгруппированная таблица *T3*, содержащая те и только те группы строк таблицы *T2*, для которых результатом вычисления *условного выражения* является *true*. *Условное выражение* *раздела HAVING* строится по синтаксическим правилам, общим для всех условных выражений, но обладает той

спецификой, что применяется к группам строк, а не к отдельным строкам. Поэтому *предикаты*, из которых строится это *условное выражение*, должны быть *предикатами* на группу в целом. В них могут использоваться имена столбцов группировки (**инварианты группы**) и так называемые *агрегатные функции* (**COUNT, SUM, MIN, MAX, AVG**) от других столбцов. Мы обсудим *агрегатные функции* более подробно в следующих лекциях.

При наличии в запросе *раздела HAVING*, которому не предшествует *раздел GROUP BY*, таблица **T1** рассматривается как сгруппированная таблица, состоящая из одной группы строк, без столбцов группирования. В этом случае *логическое выражение раздела HAVING* может состоять только из предикатов с *агрегатными функциями*, а результат вычисления этого раздела **T3** либо совпадает с таблицей **T1**, либо является пустым.

Если в операторе выборки присутствует *раздел GROUP BY*, но отсутствует *раздел HAVING*, то это трактуется как наличие раздела **HAVING true**<sup>10</sup> т. е. **T3** содержит те и только те группы строк, которые содержатся в таблице **T2**.

После выполнения *раздела WHERE* (если в запросе отсутствуют *разделы GROUP BY* и *HAVING*, случай (а)) или явно или неявно заданного *раздела HAVING* (случай (б)) выполняется *раздел SELECT*. При выполнении этого раздела на основе таблицы **T1** в случае (а) или на основе сгруппированной таблицы **T3** в случае (б) строится таблица **T4**, содержащая столько строк, сколько строк или групп строк содержится в таблицах **T1** или **T3** соответственно. Число столбцов в таблице **T4** зависит от числа элементов в списке элементов выборки (**select\_item\_commalist**) и от вида элементов.

Рассмотрим, каким образом формируются значения столбцов в таблице **T4**. Элемент списка выборки может задаваться одним из двух способов:

```
select_item ::= value_expression [[AS] column_name]
 | [correlation_name .] *
```

Сначала обсудим первый вариант. В этом случае каждый элемент списка элементов выборки соответствует столбцу таблицы **T4**. Столбцу может быть явным образом приписано имя (когда и зачем могут использоваться имена таблицы **T4**, мы обсудим позже). Порядок формирования значения этого столбца для выделенных выше случаев (а) и (б) различается, и мы рассмотрим подобные случаи по отдельности.

В случае (а) выражение, содержащееся в элементе выборки, может содержать литеральные константы и вызовы функций со значениями соответствующих типов (в том числе ниладические). Кроме того, в выражении могут использоваться имена столбцов таблицы **T1**<sup>11</sup>. Выражение вычисляется для каждой строки таблицы **T1**, и именам столбцов соответствуют значения этих столбцов в данной строке таблицы **T1**.

В случае (б), как и в случае (а), выражение, содержащееся в элементе выборки, может содержать литеральные константы и вызовы функций. Но, в отличие от случая (а), в выражении могут входить непосредственно имена только тех столбцов таблицы **T3**, которые входили в список столбцов группировки раздела **GROUP BY** оператора выборки. (Если сгруппированная таблица **T3** была образована за счет наличия *раздела HAVING* без присутствия *раздела GROUP BY*, то в выражении элемента выборки вообще нельзя непосредственно использовать имена столбцов таблицы **T3**). Имена других столбцов таблицы **T3** могут использоваться только в конструкциях вызова *агрегатных функций* **COUNT, SUM, MIN, MAX, AVG**. Выражение вычисляется для каждой группы

строк таблицы **T3**. Именам столбцов, входящих в выражение непосредственно, сопоставляются значения этих столбцов, которые соответствуют данной группе строк таблицы **T3**.

Во втором варианте спецификация элемента списка выборки вида [ **Z**. ]\* является сокращенной формой записи списка **Z.a<sub>1</sub>, Z.a<sub>2</sub>, ..., Z.a<sub>n</sub>**, где **a<sub>1</sub>, a<sub>2</sub>, ..., a<sub>n</sub>** представляет собой полный список имен столбцов таблицы, *псевдоним* которой **Z**.<sup>12</sup> Следует сделать три замечания. Во-первых, для именованной таблицы, входящей в список *раздела FROM* только один раз, можно использовать *имя таблицы* вместо псевдонима. Во-вторых, во втором варианте спецификации элемента списка выборки можно опустить *псевдоним* только в том случае, если в *разделе FROM* указана только одна таблица. В-третьих, в случае (b) второй вариант спецификации элемента выборки допустим только тогда, когда все столбцы таблицы с псевдонимом **Z** входят в список столбцов группировки *раздела GROUP BY*.

Итак, мы получили таблицу **T4**. Если в спецификации *раздела SELECT* отсутствует ключевое слово **DISTINCT**, или присутствует ключевое слово **ALL**, либо отсутствуют **ALL** и **DISTINCT**, то **T4** является результатом выполнения *раздела SELECT*. В противном случае на завершающей стадии выполнения *раздела SELECT* в таблице **T4** удаляются строки-дубликаты.

Если в операторе выборки не содержится *раздел ORDER BY*, то таблица **T4** является результирующей таблицей запроса. Иначе на завершающей стадии выполнения запроса производится сортировка строк таблицы **T4** в соответствии со списком элементов сортировки ( **order\_item\_commalist** ) *раздела ORDER BY*. В *стандарте SQL:1999* элемент списка элементов сортировки имеет следующую синтаксическую форму:

```
order_item ::= value_expression [collate_clause]
 [{ ASC | DESC }]
```

Выполнение *раздела ORDER BY* производится следующим образом<sup>13</sup> Выбирается первый элемент списка сортировки, и строки таблицы **T4** расставляются в порядке возрастания (если в элементе присутствует спецификация **ASC**; при отсутствии спецификации **ASC/DESC** предполагается наличие **ASC**) или в порядке убывания (при наличии спецификации **DESC**) в соответствии со значениями *выражения*, содержащегося в данном элементе, которые вычисляются для каждой строки таблицы **T4**. Далее выбирается второй элемент списка сортировки, и в соответствии со значениями заданного в нем *выражения* и порядка сортировки расставляются строки, которые после первого шага сортировки образовали группы с одинаковым значением *выражения* первого элемента списка сортировки. Операция продолжается до исчерпания списка элементов сортировки. Результирующий отсортированный список строк является окончательным результатом запроса.

В общем случае *выражение*, входящее в элемент списка сортировки, основывается на именах столбцов таблицы **T4** и именах столбцов таблицы, над которой вычислялся *раздел SELECT* (**T1** или **T3**). Идея состоит в том, что если некоторое *выражение* могло бы быть использовано в элементе списка выборки, то его можно использовать в элементе списка сортировки. В *стандарте SQL:1999* присутствует ряд чисто *технических ограничений* на вид *выражений*, допустимых в элементах списка сортировки, если в запросе присутствуют *разделы GROUP BY* и/или **HAVING** и если в *разделе SELECT* присутствует спецификация **DISTINCT**. Но в любом случае это *выражение* может иметь вид **a**, где **a** - имя столбца таблицы **T4**.

Заметим, что в предыдущих версиях *стандарта языка SQL*, включая SQL/92, элемент списка сортировки определялся следующим синтаксическим правилом:

```
order_item ::= { column_name | unsigned_integer }
 [{ ASC | DESC }]
```

В качестве имени столбца ( *column\_name* ) можно было использовать любое имя, вводимое для столбца таблицы **T4** в элементе списка выборки. Вместо имени столбца можно было использовать его порядковый номер ( *unsigned\_integer* ) в списке элементов выборки *раздела SELECT*. Как мы видели, в новом стандарте вторая возможность исключена. Доводом является не тот факт, что использование номеров столбцов противоречит реляционной модели. Использование номеров столбцов запрещено, поскольку не давало возможности применять в элементах списка сортировки выражения. Тем не менее, по нашему мнению, возможность использования номеров столбцов в течение долгого времени будет продолжать поддерживаться в коммерческих реализациях SQL, поскольку она применяется во многих существующих приложениях.

## Ссылки на таблицы раздела FROM

Напомним, что *раздел FROM* оператора выборки определяется синтаксическим правилом

*FROM table\_reference\_commalist*

Рассмотрим более подробно, какой вид могут иметь элементы этого списка. Для начала приведем полный набор синтаксических правил SQL:1999, определяющий *table\_reference*.<sup>14</sup>

```
table_reference ::= table_primary | joined_table
table_primary ::= table_or_query_name [[AS] correlation_name
 [(derived_column_list)]]
 | derived_table [[AS] correlation_name
 [(derived_column_list)]]
 | lateral_derived_table [[AS] correlation_name
 [(derived_column_list)]]
 | collection_derived_table [[AS] correlation_name
 [(derived_column_list)]]
 | ONLY (table_or_query_name)[[AS] correlation_name
 [(derived_column_list)]]
 | (joined_table)
table_or_query_name ::= { table_name | query_name }
derived_table ::= (query_expression)
lateral_derived_table ::= LATERAL (query_expression)
collection_derived_table ::= UNNEST
 (collection_value_exression) [WITH ORDINALITY]
```

Мы отложим до следующих лекций обсуждение *порождаемых таблиц с горизонтальной связью* ( *lateral\_derived\_table* ) и " *соединенных таблиц* " ( *joined\_table* ). Кроме того, мы не будем рассматривать в этом курсе конструкции *collection\_derived\_table* и *ONLY ( table\_or\_query\_name )*, поскольку они относятся к объектным расширениям языка SQL, которые в данном курсе подробно не

рассматриваются (на неформальном уровне объектно-реляционный подход обсуждается в последней лекции этого курса). Но даже при таких самоограничениях для дальнейшего продвижения нам придется определить несколько дополнительных синтаксических конструкций языка SQL.

### Табличное выражение, спецификация запроса и выражение запросов

*Табличным выражением* ( `table_expression` ) называется конструкция

```
table_expression ::= FROM table_reference_comma_list
 [WHERE conditional_expression]
 [GROUP BY column_name_comma_list]
 [HAVING conditional_expression]
```

*Спецификацией запроса* ( `query_specification` ) называется конструкция

```
query_specification SELECT [ALL | DISTINCT]
 select_item_comma_list table_expression
```

Наконец, *выражением запросов* ( `query_expression` ) называется конструкция

```
query_expression ::= [with_clause] query_expression_body
query_expression_body ::= { non_join_query_expression
 | joined_table }
non_join_query_expression ::= non_join_query_term
 | query_expression_body
 { UNION | EXCEPT } [ALL | DISTINCT]
 [corresponding_spec] query_term
query_term ::= non_join_query_term | joined_table
non_join_query_term ::= non_join_query_primary
 | query_term INTERSECT [ALL | DISTINCT]
 [corresponding_spec] query_primary
query_primary ::= non_join_query_primary | joined_table
non_join_query_primary ::= simple_table
 | (non_join_query_expression)
simple_table ::= query_specification
 | table_value_constructor
 | TABLE table_name
corresponding_spec ::= CORRESPONDING
 [BY column_name_comma_list]
```

Если не обращать внимания на не обсуждавшиеся пока конструкции `joined_table` и `table_value_constructor`, синтаксические правила показывают, что *выражение запросов* строится из выражений, значениями которых являются таблицы, с использованием "теоретико-множественных"<sup>15</sup> операций **UNION** (объединение), **EXCEPT** (вычитание) и **INTERSECT** (пересечение). *Операция пересечения* является "мультипликативной" и обладает более высоким приоритетом, чем "аддитивные" операции объединения и

вычитания. *Вычисление выражения* производится слева направо с учетом *приоритетов операций* и круглых скобок. При этом действуют следующие правила.

- Если *выражение запросов* не включает ни одной *теоретико-множественной операции*, то результатом вычисления *выражения запросов* является результат вычисления простой или *соединенной таблицы*.
- Если *терм* ( **non\_join\_query\_term** ) или *выражении запросов* ( **non\_join\_query\_expression** ) без соединения присутствует *теоретико-множественная операция*, то пусть **T1**, **T2** и **TR** обозначают соответственно первый операнд, второй операнд и результат терма или выражения соответственно, а **OP** - используемую *теоретико-множественную операцию*.
- Если в операции присутствует спецификация **CORRESPONDING**, то:
  1. если присутствует конструкция **BY column\_name\_comma\_list**, то все имена в этом списке должны быть различны, и каждое имя должно являться одновременно именем некоторого столбца таблицы **T1** и именем некоторого столбца таблицы **T2**, причем типы этих столбцов должны быть совместимыми; обозначим данный список имен через **SL** ;
  2. если список соответствия столбцов не задан, пусть **SL** обозначает список имен столбцов, являющихся именами столбцов и в **T1**, и в **T2**, в том порядке, в котором эти имена фигурируют в **T1** ;
  3. вычисляемые терм или *выражение запросов* без соединения эквивалентны *выражению* **(SELECT SL FROM T1) OP (SELECT SL FROM T2)**, не включающему спецификацию **CORRESPONDING**.
- При отсутствии в операции спецификации **CORRESPONDING** операция выполняется таким образом, как если бы эта спецификация присутствовала и включала конструкцию **BY column\_name\_comma\_list**, в которой были бы перечислены все столбцы таблицы **T1**.<sup>16</sup>
- При выполнении операции **OP** две строки **s1** с именами столбцов **c1, c2, ..., cn** и **s2** с именами столбцов **d1, d2, ..., dn** считаются строками-дубликатами, если для каждого **i** ( **i = 1, 2, ..., n** ) либо **ci** и **di** не содержат **NULL**, и **( ci = di ) = true**<sup>17</sup>, либо и **ci**, и **di** содержат **NULL**.
- Если в операции **OP** не задана спецификация **ALL**, то в **TR** строки-дубликаты удаляются.
- Если спецификация **ALL** задана, то пусть **s** - строка, являющаяся дубликатом некоторой строки **T1**, или некоторой строки **T2**, или обеих; пусть **m** - число дубликатов **s** в **T1**, а **n** - число дубликатов **s** в **T2**. Тогда:
  - о если указана операция **UNION**, то число дубликатов **s** в **TR** равно **m + n** ;
  - о если указана операция **EXCEPT**, то число дубликатов **s** в **TR** равно **max ((m-n),0)** ;
  - о если указана операция **INTERSECT**, то число дубликатов **s** в **TR** равно **min (m,n)**.

### Раздел **WITH** выражения запросов

Как видно из синтаксиса *выражения запросов*, в этом выражении может присутствовать раздел **WITH**. Он задается в следующем синтаксисе:

```
with_clause ::= WITH [RECURSIVE] with_element_comma_list
with_element ::= query_name [(column_name_list)]
 AS (query_expression) [search_or_cycle_clause]
```

Общую форму раздела **WITH** мы обсудим в следующих лекциях, когда будем рассматривать средства формулировки рекурсивных запросов. Пока ограничимся случаем,

когда в разделе **WITH** отсутствуют спецификация **RECURSIVE** и **search\_or\_cycle\_clause**. Тогда конструкция

**WITH query\_name (c1, c2, s cn) AS (query\_exp\_1) query\_exp\_2**

означает, что в любом месте *выражения запросов* **query\_exp\_2**, где допускается появление ссылки на таблицу, можно использовать имя **query\_name**. Можно считать, что перед выполнением **query\_exp\_2** происходит выполнение **query\_exp\_1**, и результирующая таблица с именами столбцов **c1, c2, ... cn** сохраняется под именем **query\_name**. Как мы увидим позже, в этом случае раздел **WITH** фактически служит для локального определения представляемой таблицы ( **VIEW** ).

### Конструкторы значения строки и таблицы

Чтобы завершить обсуждение *выражений запросов* (с учетом того, что конструкция *соединенных таблиц* ( **joined\_table** ) отложена на следующие лекции), нам осталось рассмотреть конструкции **table\_value\_constructor** и **TABLE table\_name**.

В определении *конструктора значения-таблицы* используется *конструктор значения-строки*, который строит упорядоченный набор скалярных значений, представляющий строку (возможно и использование подзапроса<sup>18</sup>):

```
row_value_constructor ::= row_value_constructor_element
 | [ROW] (row_value_constructor_element_comma_list)
 | row_subquery
row_value_constructor_element ::= value_expression | NULL | DEFAULT
```

Заметим, что значение элемента по умолчанию можно использовать только в том случае, когда *конструктор значения-строки* применяется в операторе **INSERT** (тогда этим значением будет значение по умолчанию соответствующего столбца).

*Конструктор значения-таблицы* производит таблицу на основе заданного набора *конструкторов значений-строк*:

```
table_value_constructor ::= VALUES
 row_value_constructor_comma_list
```

Конечно, для того чтобы корректно построить таблицу, требуется, чтобы строки, производимые всеми конструкторами строк, были одной и той же степени и чтобы типы (или домены) соответствующих столбцов являлись приводимыми.

Наконец, конструкция **TABLE table\_name** является сокращенной формой записи выражения **SELECT \* FROM table\_name**.

### Ссылки на базовые, представляемые и порождаемые таблицы

Теперь мы можем завершить обсуждение разновидностей ссылок на таблицу в разделе **FROM**. Для удобства повторим синтаксические правила (опустив конструкции, рассмотрение которых отложено на следующие лекции или выходит за пределы материала данного курса):

```

table_reference ::= table_primary
table_primary ::= table_or_query_name [[AS] correlation_name
 [(derived_column_list)]]
 | derived_table [AS] correlation_name
 [(derived_column_list)]
table_or_query_name ::= { table_name | query_name }
derived_table ::= (query_expression)

```

Итак, в самом простом случае в качестве ссылки на таблицу используется *имя таблицы* (базовой или представляемой) или имя запроса, присоединенного к данному запросу с помощью раздела **WITH**. В другом случае ( **derived\_table** ) порождаемая таблица задается *выражением запроса*, заключенным в круглые скобки. Явное указание имен столбцов результата запроса из раздела **WITH** или порождаемой таблицы требуется в том случае, когда эти имена не выводятся явно из соответствующего *выражения запроса*. Обратите внимание, что в таких случаях в соответствующем элементе списка *раздела FROM* должен указываться *псевдоним* (**correlation\_name** ), потому что иначе таблица была бы вообще лишена имени. Можно считать, что *выражение запроса* вычисляется и сохраняется во временной таблице при обработке *раздела FROM*.

Возможно, некоторых читателей смутила рекурсивная природа синтаксических определений, приведенных в этом подразделе. Чтобы определить понятие ссылки на таблицу в *разделе FROM* оператора выборки, который опирается на *спецификацию запроса*, нам пришлось ввести более общее понятие *выражения запросов*, в определении которого используется *спецификация запроса*. Да, действительно, многие синтаксические конструкции SQL определяются рекурсивно. Но эта рекурсия никогда не приводит к заикливанию. В частности, раскрытка рекурсии операторов выборки основывается на базовой, не выделяемой отдельными синтаксическими правилами форме, в которой в *разделе FROM* указываются только имена базовых таблиц.

## Представляемые таблицы, или представления (VIEW)

Еще одним примером рекурсивности спецификаций языка SQL является то, что в конце этой лекции мы вынуждены прервать обсуждение оператора выборки и ввести понятие *представляемой таблицы, или представления*, которую можно использовать в операторе выборки наряду с *базовыми таблицами*. Только после этого можно будет считать обсуждение ссылок на таблицы в *разделе FROM* условно завершенным. Итак, оператор создания представления в общем случае определяется следующими синтаксическими правилами:

```

create_view ::= CREATE [RECURSIVE] VIEW table_name
 [column_name_comma_list]
 AS query_expression
 [WITH [CASCADED | LOCAL] CHECK OPTION]

```

Рекурсивные представления (такие, в определении которых присутствует ключевое слово **RECURSIVE** ) и необязательный раздел **WITH CHECK OPTION** мы обсудим в следующих лекциях (пока лишь заметим, что этот раздел связан с особенностями выполнения операций обновления базы данных через представления). Здесь мы кратко рассмотрим только простую форму представлений, определяемых по следующим правилам:

```
create_view ::= CREATE VIEW table_name
 [column_name_comma_list]
 AS query_expression
```

*Имя таблицы*, задаваемое в определении представления, существует в том же пространстве имен, что и имена базовых таблиц, и, следовательно, должно отличаться от всех имен таблиц (базовых и представляемых), созданных тем же пользователем. Если имя представления встречается в разделе *FROM* какого-либо оператора выборки, то вычисляется *выражение запроса*, указанное в разделе **AS**, и оператор выборки работает с результирующей таблицей этого *выражения запроса*.<sup>19</sup> Явное указание имен столбцов представляемой таблицы требуется в том случае, когда эти имена не выводятся из соответствующего *выражения запроса*.

Как и для всех других вариантов оператора **CREATE**, для **CREATE VIEW** имеется обратный оператор **DROP VIEW table\_name**, выполнение которого приводит к отмене определения представления (реально это выражается в удалении данных о представлении из таблиц-каталогов базы данных). После выполнения операции пользоваться представлением с данным именем становится невозможно<sup>20</sup>

## Заключение

В ходе чтения лекций, посвященных *оператору SELECT* языка *SQL*, мне неоднократно случалось слышать жалобы студентов на сухость начального материала и отсутствие иллюстрирующих примеров. Однако я не встречал ни одного учебного пособия по языку *SQL*, основанного на примерах (среди многочисленных изданий типа "*SQL* за 24 часа", "*SQL* для чайников" и даже "*SQL* для идиотов"), который действительно давал бы *представление* об *SQL* как языке, а не служил инструкцией армейского типа.

Сложность организации оператора выборки не позволяет сразу начинать с полноценных примеров, а для демонстрации примеров промежуточных конструкций требуется создание неприемлемо громоздкого контекста. Поэтому могу лишь принести извинения за некоторую сухость этой лекции.

С другой стороны, теперь мы уже вплотную подошли к тому этапу, на котором возможно использование иллюстраций, и в следующих лекциях их будет достаточно, хотя проиллюстрировать все интересные разновидности *оператора SELECT* все равно не представляется возможным, поскольку число вариантов близко к астрономическому.

## Лабораторная работа 4. Предикаты раздела WHERE оператора SELECT

### Введение

Конструкции оператора **SELECT** языка *SQL* в значительной степени ортогональны. В частности, выбор способа указания ссылки на таблицы в разделе **FROM** никак не влияет на выбор варианта формирования условия выборки в разделе **WHERE**. Это полезное свойство языка позволяет нам абстрагироваться от обсуждавшегося в предыдущей лекции многообразия способов указания ссылки на таблицу и сосредоточиться на возможностях формирования запросов при использовании различных *предикатов*, допускаемых *стандартом SQL:1999* в логических выражениях раздела *WHERE*.

В стандарте *SQL:1999* специфицированы 12 разновидностей *предикатов*, причем некоторые из них в действительности представляют собой семейства (например, под общим названием *предиката сравнения* скрываются шесть видов *предикатов*). Набор допустимых *предикатов* в *SQL* явно избыточен, но тем не менее в языке *SQL* имеется явная тенденция расширения этого набора. В частности, в *SQL:2003* в связи с введением генератора *типов мультимножеств* в дополнение ко всем разновидностям *предикатов SQL:1999* появилось три новых вида *предикатов*: *предикаты* для проверки того, что заданное значение является элементом *мультимножества* (*MEMBER*); что одно *мультимножество* входит в другое *мультимножество* (*SUBMULTISET*) и что *мультимножество* не содержит дубликаты (*IS A SET*). В этом курсе мы не приводим подробного описания этих видов *предикатов* по нескольким причинам:

- введение *конструктора типов мультимножеств* в стандарте *SQL:2003* не означает, что достигнута общая цель разработчиков стандарта *SQL* по обеспечению полного набора *типов коллекций*; по всей видимости, в будущих версиях стандарта появятся дополнительные *конструкторы типов коллекций*, и набор видов *предикатов* изменится;
- *предикаты* с *мультимножествами* трудно пояснять и иллюстрировать в отрыве от других объектно-реляционных средств языка *SQL*;
- включение подобного материала в данную лекцию заметно увеличило бы ее объем и затруднило понимание более традиционных конструкций.

В лекции содержится много примеров запросов с использованием различных видов *предикатов*. Для полного усвоения материала требуется тщательно проанализировать эти примеры.

## Логические выражения раздела WHERE

Синтаксически *логическое выражение раздела WHERE* определяется как *булевское выражение* (*boolean\_value\_expression*), правила построения которого обсуждались в предыдущей лекции. Основой *логического выражения* являются *предикаты*. *Предикат* позволяет специфицировать условие, результатом вычисления которого может быть *true*, *false* или *unknown*. В языке *SQL:1999* допустимы следующие *предикаты*:<sup>1</sup>

```
predicate ::= comparison_predicate
| between_predicate
| null_predicate
| in_predicate
| like_predicate
| similar_predicate
| exists_predicate
| unique_predicate
| overlaps_predicate
| quantified_comparison_predicate
| match_predicate
| distinct_predicate
```

Далее мы будем последовательно обсуждать разные виды *предикатов* и приводить примеры запросов с использованием базы данных **СЛУЖАЩИЕ-ОТДЕЛЫ-ПРОЕКТЫ**,

определения таблиц которой на языке *SQL* были приведены в лекции 12. Для удобства повторим здесь структуру таблиц.

*EMP:*

EMP\_NO : EMP\_NO

EMP\_NAME : VARCHAR

EMP\_BDATE : DATE

EMP\_SAL : SALARY

DEPT\_NO : DEPT\_NO

PRO\_NO : PRO\_NO

*DEPT:*

DEPT\_NO : DEPT\_NO

DEPT\_NAME : VARCHAR

DEPT\_EMP\_NO : INTEGER

DEPT\_TOTAL\_SAL : SALARY

DEPT\_MNG : EMP\_NO

*PRO:*

PRO\_NO : PRO\_NO

PRO\_TITLE : VARCHAR

PRO\_SDATE : DATEP

PRO\_DURAT : INTERVAL

PRO\_MNG : EMP\_NO

PRO\_DESC : CLOB

Столбцы **EMP\_NO**, **DEPT\_NO** и **PRO\_NO** являются *первичными ключами* таблиц **EMP**, **DEPT** и **PRO** соответственно.

Столбцы **DEPT\_NO** и **PRO\_NO** таблицы **EMP** являются *внешними ключами*, ссылающимися на таблицы **DEPT** и **PRO** соответственно ( **DEPT\_NO** указывает на отделы, в которых работают служащие, а **PRO\_NO** - на проекты, в которых они участвуют; оба столбца могут принимать неопределенные значения). Столбец **DEPT\_MNG** является *внешним ключом* таблицы **DEPT** ( **DEPT\_MNG** указывает на служащих, которые исполняют обязанности руководителей отделов; у отдела может не быть руководителя, и один служащий не может быть руководителем двух или более отделов). Столбец **PRO\_MNG** является *внешним ключом* таблицы **PRO** ( **PRO\_MNG** указывает на служащих, которые являются *менеджерами проектов*, у проекта всегда есть *менеджер*, и один служащий не может быть менеджером двух или более проектов).

## Предикат сравнения

Этот *предикат* предназначен для спецификации сравнения двух строчных значений. Синтаксис *предиката* следующий:

comparison\_predicate ::=

row\_value\_constructor comp\_op row\_value\_constructor

comp\_op ::= | < > ("неравно") | < | >

| <= "меньше или равно" | >= "больше или равно"

Строки, являющиеся операндами операции сравнения, должны быть одинаковой степени. Типы данных соответствующих значений строк-операндов должны быть совместимы.

Пусть **X** и **Y** обозначают соответствующие элементы строк-операндов, а **xv** и **yv** - их значения. Тогда:

1. если **xv** и/или **yv** являются неопределенными значениями, то значение условия **X comp\_op Y** - **unknown** ;
2. в противном случае значением условия **X comp\_op Y** является **true** или **false** в соответствии с естественными правилами применения операции сравнения.

При этом:

- Числа сравниваются в соответствии с правилами *алгебры*.

- Сравнение двух символьных строк производится следующим образом:
  - о если длина строки **X** не равна длине строки **Y**, то для выравнивания длин строк более короткая строка расширяется символами набивки (**pad symbol**); если для используемого набора символов порядок сортировки явным образом не специфицирован, то в качестве символа набивки используется пробел;
  - о далее производится лексикографическое сравнение строк в соответствии с предопределенным или явно определенным порядком сортировки символов.
- Сравнение двух битовых строк **X** и **Y** основано на сравнении соответствующих бит. Если **X<sub>i</sub>** и **Y<sub>i</sub>** - значения **i**-тых бит **X** и **Y** соответственно и если **lx** и **ly** обозначает длину в битах **X** и **Y** соответственно, то:
  - о **X** равно **Y** тогда и только тогда, когда **lx = ly** и **X<sub>i</sub> = Y<sub>i</sub>** для всех **i**;
  - о **X** меньше **Y** тогда и только тогда, когда (a) **lx < ly** и **X<sub>i</sub> = Y<sub>i</sub>** для всех **i** меньших или равных **lx**, или (b) **X<sub>i</sub> = Y<sub>i</sub>** для всех **i < n** и **X<sub>n</sub> = 0**, а **Y<sub>n</sub> = 1** для некоторого **n** меньшего или равного **min (lx, ly)**.
- Сравнение двух значений типа дата-время производится в соответствии с видом интервала, который получается при вычитании второго значения из первого. Пусть **X** и **Y** - сравниваемые значения, а **H** - наименее значимое поле даты-времени **X** и **Y**. Результат сравнения **X comp\_op Y** определяется как **(X - Y) H comp\_op INTERVAL (0) H**. (Два значения типа дата-время сравнимы только в том случае, если они содержат одинаковый набор полей даты-времени.)
- Сравнение двух значений анонимного строкового типа производится следующим образом. Пусть **Rx** и **Ry** обозначают строки-операнды, а **Rx<sub>i</sub>** и **Ry<sub>i</sub>** - **i**-тые элементы **Rx** и **Ry** соответственно. Вот как определяется результат сравнения **Rx comp\_op Ry**:
  - о **Rx = Ry** есть **true** тогда и только тогда, когда **Rx<sub>i</sub> = Ry<sub>i</sub>** есть **true** для всех **i**;
  - о **Rx <> Ry** есть **true** тогда и только тогда, когда **Rx<sub>i</sub> <> Ry<sub>i</sub>** есть **true** для некоторого **i**;
  - о **Rx < Ry** есть **true** тогда и только тогда, когда **Rx<sub>i</sub> = Ry<sub>i</sub>** есть **true** для всех **i < n**, и **Rx<sub>n</sub> < Ry<sub>n</sub>** есть **true** для некоторого **n**;
  - о **Rx > Ry** есть **true** тогда и только тогда, когда **Rx<sub>i</sub> = Ry<sub>i</sub>** есть **true** для всех **i < n**, и **Rx<sub>n</sub> > Ry<sub>n</sub>** есть **true** для некоторого **n**;
  - о **Rx <= Ry** есть **true** тогда и только тогда, когда **Rx = Ry** есть **true** или **Rx < Ry** есть **true**;
  - о **Rx >= Ry** есть **true** тогда и только тогда, когда **Rx = Ry** есть **true** или **Rx > Ry** есть **true**;
  - о **Rx = Ry** есть **false** тогда и только тогда, когда **Rx <> Ry** есть **true**;
  - о **Rx <> Ry** есть **false** тогда и только тогда, когда **Rx = Ry** есть **true**;
  - о **Rx < Ry** есть **false** тогда и только тогда, когда **Rx >= Ry** есть **true**;
  - о **Rx > Ry** есть **false** тогда и только тогда, когда **Rx <= Ry** есть **true**;
  - о **Rx <= Ry** есть **false** тогда и только тогда, когда **Rx > Ry** есть **true**;
  - о **Rx >= Ry** есть **false** тогда и только тогда, когда **Rx < Ry** есть **true**;
  - о **Rx comp\_op Ry** есть **unknown** тогда и только тогда, когда **Rx comp\_op Ry** не есть **true** или **false**.

#### Примеры запросов с использованием предиката сравнения

```
SELECT DISTINCT EMP.DEPT_NO
FROM EMP
```

```
WHERE EMP.EMP_NAME = 'Smith';
```

14.1. Найти номера отделов, в которых работают служащие с фамилией 'Smith'.

Мы добавили спецификацию **DISTINCT** в раздел **SELECT**, потому что в одном отделе могут работать несколько служащих с фамилией 'Smith', а их число нас в данном случае не интересует. Кстати, если бы нас интересовало число служащих с фамилией 'Smith' в

каждом отделе, где такие служащие работают, то следовало бы, например, написать такой запрос ( [пример 14.1.1](#)):

```
SELECT EMP.DEPT_NO, COUNT(*)
FROM EMP
WHERE EMP.NAME = 'Smith'
GROUP BY EMP.DEPT_NO;
```

#### 14.1.1.

В этом варианте запроса спецификация *DISTINCT* не требуется, поскольку в запросе содержится раздел **GROUP BY**, группировка производится в соответствии со значениями столбца *EMP.DEPT\_NO*, и строка результата соответствует одной группе.

```
SELECT EMP.EMP_NO, EMP.EMP_NAME, EMP.DEPT_NO
FROM EMP
WHERE EMP.EMP_BDATE > DATE '1965-04-15';
```

#### 14.2. Найти номера, имена и номера отделов служащих, родившихся после 15 апреля 1965 г.

В результате этого запроса дубликатов быть не может, поскольку в список выборки включен столбец, являющийся первичным ключом таблицы *EMP*. Должно быть ясно, что по этой причине все строки результата будут различными.

```
SELECT EMP.EMP_NO, EMP.EMP_NAME, EMP.DEPT_NO
FROM EMP
WHERE EMP.EMP_SAL > 0.1 *
 (SELECT DEPT_TOTAL_SAL
 FROM DEPT
 WHERE DEPT.DEPT_NO = EMP.DEPT_NO);
```

#### 14.3. Найти номера, имена и номера отделов служащих, размер заработной платы которых составляет больше одной десятой объема фонда заработной платы их отделов.

В этом *SQL-запросе* имеются две интересные особенности, которые мы до сих пор не обсуждали. Во-первых, второй операнд операции сравнения содержит *подзапрос*, возвращающий единственное значение, поскольку *логическое выражение* раздела **WHERE** этого *подзапроса* состоит из условия, однозначно определяющего значение *первичного* ключа таблицы *DEPT*. Во-вторых, в условии раздела **WHERE** *подзапроса* используется ссылка на столбец таблицы *EMP*, указанной в разделе **FROM** "внешнего" запроса. Подобные *подзапросы* в терминологии SQL традиционно называются *корреляционными*, и их следует понимать следующим образом<sup>2</sup>.

При выполнении внешнего запроса последовательно, строка за строкой, в некотором порядке, определяемом системой, производится проверка соответствия строк результирующей таблицы раздела **FROM** условию раздела **WHERE**. Если это условие включает *корреляционные подзапросы*, то внутри каждого из этих подзапросов ссылка на столбец внешней таблицы трактуется как ссылка на столбец текущей строки данной таблицы во внешнем цикле. Естественно, условие **WHERE** любого *подзапроса* может включать более глубоко вложенные *подзапросы*, на которые распространяется то же правило корреляции с внешними таблицами.

Кстати, эквивалентная формулировка на языке SQL [примера 14.3](#) выглядит следующим образом ( [пример 14.3.1](#)):

```
SELECT EMP.EMP_NO, EMP.EMP_NAME, EMP.DEPT_NO
FROM EMP, DEPT
WHERE EMP.DEPT_NO = DEPT.DEPT_NO AND
 EMP.EMP_SAL > 0.1 * DEPT.TOTAL_SAL;
```

14.3.1.

Мы видим, что в терминах *реляционной алгебры* этот запрос представляет собой ограничение (по условию  $EMP.EMP\_SAL > 0.1 * DEPT.TOTAL\_SAL$ ) *эквисоединения* таблиц *EMP* и *DEPT* (по условию  $EMP.DEPT\_NO = DEPT.DEPT\_NO$ ). Подобную операцию часто называют *полусоединением (semijoin)*, поскольку в результирующей таблице используются столбцы только одного из операндов операции эквисоединения. Мы привели вторую формулировку запроса, преследуя две цели: (1) продемонстрировать, каким образом *предикат сравнения* можно использовать для задания условия соединения, и (2) показать, что запросы, содержащие вложенные запросы, часто могут быть переформулированы в запросы с соединениями.

```
SELECT EMP1.EMP_NO, EMP1.EMP_NAME,
 EMP1.DEPT_NO, EMP2.EMP_NAME
FROM EMP AS EMP1, EMP AS EMP2, DEPT
WHERE EMP1.EMP_SAL < 15000.00 AND
 EMP1.DEPT_NO = DEPT.DEPT_NO AND
 DEPT.DEPT_MNG = EMP2.EMP_NO;
```

14.4. Найти номера, имена, номера отделов и имена руководителей отделов служащих, размер заработной платы которых меньше 15000 руб.

Этот запрос представляет собой эквисоединение ограничения таблицы *EMP* (по условию  $EMP\_SAL < 15000.00$ ) с таблицами *DEPT* и *EMP* (по условиям  $EMP.DEPT\_NO = DEPT.DEPT\_NO$  и  $DEPT.DEPT\_MNG = EMP2.EMP\_NO$  соответственно). Таблица *EMP* участвует в качестве операнда операции эквисоединения два раза. Поэтому в разделе *FROM* ей присвоены два псевдонима - *EMP1* и *EMP2*. Следуя предписанному стандартom порядку выполнения запроса, можно считать, что введение этих псевдонимов обеспечивает переименование столбцов таблицы *EMP*, требуемое для выполнения раздела *FROM* с образованием расширенного *декартова произведения* таблиц-операндов.<sup>3</sup> Заметим также, что в данном случае мы имеем дело с *полным эквисоединением* трех таблиц (а не с *полусоединением*), поскольку в списке выборки присутствуют имена столбцов каждой из них.

Покажем способ формулировки этого запроса с использованием вложенного *подзапроса* в качестве элемента списка выборки ( [пример 14.4.1](#)):

```
SELECT EMP.EMP_NO, EMP.EMP_NAME, EMP.DEPT_NO,
 (SELECT EMP_NAME
 FROM EMP
 WHERE EMP_NO = DEPT_MNG)
FROM EMP, DEPT
WHERE EMP.EMP_SAL < 15000.00 AND
 EMP.DEPT_NO = DEPT.DEPT_NO;
```

#### 14.4.1.

Как показывает последний пример, в условии выборки *подзапроса*, участвующего в списке выборки, можно использовать имена столбцов таблиц внешнего запроса. Из этой возможности языка SQL видно, что в разделе "Общие синтаксические правила построения *скалярных выражений*" предыдущей лекции для облегчения понимания материала мы немного исказили семантику оператора выборки. Там было сказано следующее: "После выполнения раздела **WHERE** (если в запросе отсутствуют разделы **GROUP BY** и **HAVING**, случай (a)) или выполнения явно или неявно заданного раздела **HAVING** (случай (b)) выполняется раздел **SELECT**. При выполнении этого раздела на основе таблицы **T1** в случае (a) или на основе сгруппированной таблицы **T3** в случае (b) строится таблица **T4**, содержащая столько строк, сколько строк или групп строк содержится в таблицах **T1** или **T3** соответственно". В действительности, в общем случае очередная строка таблицы **T4** должна строиться в тот момент, когда очередная строка или группа строк заносится в таблицу **T1** или **T3** соответственно.

#### Предикат between

*Предикат* позволяет специфицировать условие вхождения в диапазон значений. Операндами являются строки:

```
between_predicate ::=
 row_value_constructor [NOT] BETWEEN
 row_value_constructor AND row_value_constructor
```

Все три строки-операнды должны иметь одну и ту же степень. Типы данных соответствующих значений строк-операндов должны быть совместимыми.

Пусть **X**, **Y** и **Z** обозначают первый, второй и третий операнды. Тогда по определению выражение **X NOT BETWEEN Y AND Z** эквивалентно выражению **NOT (X BETWEEN Y AND Z)**. Выражение **X BETWEEN Y AND Z** по определению эквивалентно булевскому выражению **X >= Y AND X <= Z**.

#### Примеры запросов с использованием предиката between

```
SELECT EMP_NO, EMP_NAME, EMP_SAL
FROM EMP
```

```
WHERE EMP_SAL BETWEEN 12000.00 AND 15000.00;
```

14.5. Найти номера, имена и размер зарплаты служащих, получающих зарплату в размере от 12000 до 15000 руб.

```
SELECT EMP_NO, EMP_NAME, EMP_SAL
FROM EMP
WHERE EMP_SAL BETWEEN
 (SELECT AVG(EMP1.EMP_SAL)
 FROM EMP EMP1
 WHERE EMP.DEPT_NO = EMP1.DEPT_NO)
 AND
 (SELECT EMP1.EMP_SAL
 FROM EMP EMP1
 WHERE EMP1.EMP_NO =
 (SELECT DEPT.DEPT_MNG
```

FROM DEPT  
WHERE DEPT.DEPT\_NO = EMP.DEPT\_NO));

14.6. Найти номера, имена и размер зарплаты служащих, получающих зарплату, размер которой не меньше средней зарплаты служащих своего отдела и не больше зарплаты руководителя отдела.

В этом запросе можно выделить три интересных момента. Во-первых, диапазон значений предиката *BETWEEN* задан двумя подзапросами, результатом каждого из которых является единственное значение. Первый *подзапрос* выдает единственное значение, поскольку в списке выборки содержится *агрегатная функция* (*AVG*) и отсутствует раздел *GROUP BY*, а второй - потому что в его разделе *WHERE* присутствует условие, задающее единственное значение *первичного ключа*. Во-вторых, в обоих подзапросах таблица *EMP* получает *псевдоним EMP1* (в формулировке этого запроса мы старались использовать как можно меньше вспомогательных идентификаторов). Поскольку *подзапросы* выполняются независимо один от другого, использование общего имени не вызывает проблем. Наконец, в условии второго *подзапроса* присутствует более глубоко вложенный *подзапрос*, и в условии его раздела *WHERE* используется ссылка на столбец таблицы из самого внешнего раздела *FROM*.

### Предикат is null

Предикат *is null* позволяет проверить, являются ли неопределенными значения всех элементов строки-операнда:

*null\_predicate ::= row\_value\_constructor IS [ NOT ] NULL*

Пусть *X* обозначает строку-операнд. Если значения всех элементов *X* являются неопределенными, то значением условия *X IS NULL* является *true*; иначе - *false*. Если ни у одного элемента *X* значение не является неопределенным, то значением условия *X IS NOT NULL* является *true*; иначе - *false*.

Замечание: условие *X IS NOT NULL* имеет то же значение, что условие *NOT X IS NULL* для любого *X* в том и только в том случае, когда степень *X* равна 1. Полная семантика предиката *null* приведена в [таблице 14.1](#).

Таблица 14.1.

| Вид операнда                               | Вид условия      |                      |                      |                          |
|--------------------------------------------|------------------|----------------------|----------------------|--------------------------|
|                                            | <i>X IS NULL</i> | <i>X IS NOT NULL</i> | <i>NOT X IS NULL</i> | <i>NOT X IS NOT NULL</i> |
| Степень 1: значение <i>NULL</i>            | true             | false                | false                | true                     |
| Степень 1: значение отлично от <i>NULL</i> | false            | true                 | true                 | false                    |

|                                                                    |       |       |       |       |
|--------------------------------------------------------------------|-------|-------|-------|-------|
| Степень > 1: у всех элементов значение <b>NULL</b>                 | true  | false | false | true  |
| Степень > 1: у некоторых(не у всех) элементов значение <b>NULL</b> | false | false | true  | true  |
| Степень > 1: ни у одного элемента нет значения <b>NULL</b>         | false | true  | true  | false |

### Примеры запросов с использованием предиката null

На самом деле, в нашей формулировке запроса из [примера 14.6](#) есть одна неточность. Если у некоторого служащего номер отдела неизвестен (значение столбца **EMP.DEPT\_NO** у соответствующей строки таблицы служащих является неопределенным), то бессмысленно вычислять средний размер зарплаты отдела этого служащего и находить размер зарплаты руководителя отдела. Формулировка из [примера 14.6](#) приведет к правильному результату, но это неочевидно<sup>4</sup>. Чтобы сделать формулировку более понятной (и, возможно, помочь системе выполнить запрос более эффективно), нужно воспользоваться предикатом **IS NOT NULL** и переписать запрос следующим образом:

```
SELECT EMP_NO, EMP_NAME, EMP_SAL
FROM EMP
WHERE DEPT_NO IS NOT NULL AND
 EMP_SAL BETWEEN
 (SELECT AVG(EMP1.EMP_SAL)
 FROM EMP EMP1
 WHERE EMP.DEPT_NO = EMP1.DEPT_NO)
 AND
 (SELECT EMP1.EMP_SAL
 FROM EMP EMP1
 WHERE EMP1.EMP_NO =
 (SELECT DEPT.DEPT_MNG
 FROM DEPT
 WHERE DEPT.DEPT_NO = EMP.DEPT_NO));
```

14.7.

```
SELECT EMP_NO, EMP_NAME
FROM EMP
WHERE DEPT_NO IS NULL;
```

14.8. Найти номера и имена служащих, номер отдела которых неизвестен.

### Предикат in

*Предикат* позволяет специфицировать условие вхождения строчного значения в указанное множество значений. Синтаксические правила следующие:

```
in_predicate ::= row_value_constructor [NOT]
 IN in_predicate_value
```

```
in_predicate_value ::= table_subquery
| (value_expression_comma_list)
```

Строка, являющаяся первым операндом, и таблица-второй операнд должны быть одинаковой степени. В частности, если второй операнд представляет собой список значений, то первый операнд должен иметь степень 1. Типы данных соответствующих столбцов операндов должны быть совместимы.

Пусть **X** обозначает строку-первый операнд, а **S** - множество строк второго операнда. Обозначим через **s** строку-элемент этого множества. Тогда по определению условие **X IN S** эквивалентно булевскому выражению **OR<sub>sS</sub> (X = s)**. Другими словами, **X IN S** принимает значение **true** в том и только в том случае, когда во множестве **S** существует хотя бы один элемент **s**, такой, что значением предиката **X = s** является **true**. **X IN S** принимает значение **false** в том и только том случае, когда для всех элементов **s** множества **S** значением операции сравнения **X = s** является **false**. Иначе значением условия **X IN S** является **unknown**. Заметим, что для пустого множества **S** значением **X IN S** является **false**.

По определению условие **X NOT IN S** эквивалентно **NOT (X IN S)**.

#### Примеры запросов с использованием предиката **in**

```
SELECT EMP_NO, EMP_NAME, DEPT_NO
FROM EMP
WHERE DEPT_NO IN (15, 17, 19);
```

14.9. Найти номера, имена и номера отделов служащих, работающих в отделах 15, 17 и 19.

Конечно, эта формулировка запроса эквивалентна следующей формулировке ([пример 14.9.1](#)):

```
SELECT EMP_NO, EMP_NAME, DEPT_NO
FROM EMP
WHERE DEPT_NO = 15
OR DEPT_NO = 17
OR DEPT_NO = 19;
14.9.1.
SELECT EMP_NO
FROM EMP
WHERE EMP_NO NOT IN (SELECT DEPT_MNG FROM DEPT)
AND EMP_SAL IN (SELECT EMP_SAL FROM EMP,
DEPT WHERE EMP_NO = DEPT_MNG);
```

14.10. Найти номера служащих, не являющихся руководителями отделов и получающих зарплату, размер которой равен размеру зарплаты какого-либо руководителя отдела.

Запросы, содержащие предикат **IN** с *подзапросом*, легко переформулировать в запросы с соединениями. Например, запрос из [примера 14.10](#) эквивалентен следующему запросу с соединениями ([пример 14.10.1](#)):

```
SELECT DISTINCT EMP_NO
FROM EMP, EMP EMP1, DEPT
```

```
WHERE EMP_NO NOT IN (SELECT DEPT_MNG FROM DEPT)
AND EMP_SAL = EMP1_SAL
AND EMP1.EMP_NO = DEPT.DEPT_MNG;
14.10.1.
```

По поводу этой второй формулировки следует сделать два замечания. Во-первых, как видно, мы изменили только ту часть условия, в которой использовался предикат *IN*, и не затронули предикат *NOT IN*. Запросы с предикатами *NOT IN* запросами с соединениями так просто не заменяются. Во-вторых, в разделе *SELECT* было добавлено ключевое слово *DISTINCT*, потому что в результате запроса во второй формулировке для каждого служащего будет содержаться столько строк, сколько существует руководителей отделов, получающих такую же зарплату, что и данный служащий.

### Предикат like

Формально предикат *like* определяется следующими синтаксическими правилами:

```
like_predicate ::= source_value [NOT]
LIKE pattern_value [ESCAPE escape_value]
source_value ::= value_expression
pattern_value ::= value_expression
escape_value ::= value_expression
```

Все три операнда ( *source\_value*, *pattern\_value* и *escape\_value* ) должны быть одного типа: либо *типа символьных строк*, либо типа битовых строк<sup>5</sup>. В первом случае значением последнего операнда должна быть строка из одного символа, во втором - строка из 8 бит. Второй операнд, как правило, задается литералом соответствующего типа. В обоих случаях значение *предиката* равняется *true* в том и только в том случае, когда исходная строка ( *source\_value* ) может быть сопоставлена с заданным шаблоном ( *pattern\_value* ).

Если обрабатываются *символьные строки*, и если раздел *ESCAPE* условия отсутствует, то при сопоставлении шаблона со строкой производится специальная интерпретация двух символов шаблона: символ подчеркивания ( *'\_'* ) обозначает любой одиночный символ; символ процента ( *'%'* ) обозначает последовательность произвольных символов произвольной длины (длина последовательности может быть нулевой). Если же раздел *ESCAPE* присутствует и специфицирует некоторый одиночный символ *x*, то пары символов " *x\_* " и " *x%* " представляют одиночные символы " *\_* " и " *%* " соответственно.

В случае обработки битовых строк сопоставление шаблона со строкой производится восьмерками соседних бит ( *октетами* ). В соответствии со *стандартом SQL:1999*, при сопоставлении шаблона со строкой производится специальная интерпретация *октетов* со значениями *X'25'* и *X'5F'* (коды символов подчеркивания и процента в кодировке ASCII). Первый *октет* обозначает любой одиночный *октет*, а второй - последовательность произвольной длины произвольных *октетов* (длина может быть нулевой). В разделе *ESCAPE* указывается *октет*, отменяющий специальную интерпретацию *октетов X'25'* и *X'5F'*.

Значение предиката *like* есть *unknown*, если значение первого или второго операндов является неопределенным. Условие *x NOT LIKE y ESCAPE z* эквивалентно условию *NOT x LIKE y ESCAPE z*.

### Примеры запросов с использованием предиката like

```
SELECT PRO_TITLE
FROM PRO
WHERE PRO_TITLE LIKE '%next%step%'
 OR PRO_TITLE LIKE 'Next%step%';
```

14.11. Найти номера проектов, в названии которых присутствуют слова 'next' и 'step'. Слова должны следовать именно в такой последовательности, но слово 'next' может быть первым в названии проекта.

Это очень неудачный запрос, потому что его выполнение, скорее всего, вынудит СУБД просмотреть все строки таблицы *PRO* и для каждой строки выполнить две проверки столбца *PRO\_TITLE*. Можно немного улучшить формулировку с небольшим риском получить неверный ответ ( [пример 14.11.1](#)):

```
SELECT PRO_TITLE
FROM PRO
WHERE PRO_TITLE LIKE '%ext%step%';
14.11.1.
SELECT DISTINCT DEPT.DEPT_NO
FROM EMP, DEPT, PRO
WHERE EMP.EMP_NO = PRO.PRO_MNG
 AND EMP.DEPT_NO = DEPT.DEPT_NO
 AND PRO.PRO_TITLE LIKE DEPT.DEPT_NAME || '%';
```

14.12. Найти номера отделов, служащие которых являются менеджерами проектов, и название каждого из этих проектов начинается с названия отдела.

Вот как может выглядеть формулировка этого запроса, если использовать вложенные *подзапросы* ( [пример 14.12.1](#)):

```
SELECT DEPT.DEPT_NO
FROM DEPT
WHERE DEPT.DEPT_NO IN
 (SELECT EMP.DEPT_NO
 FROM EMP
 WHERE EMP.EMP_NO IN
 (SELECT PRO.PRO_MNG FROM PRO
 WHERE PRO.PRO_TITLE LIKE DEPT.DEPT_NAME || '%'));
```

14.12.1.

```
SELECT DEPT_NO
FROM DEPT
WHERE DEPT_NAME NOT LIKE 'Software%';
```

14.13. Найти номера отделов, названия которых не начинаются со слова 'Software'.

### Предикат similar

Формально предикат *similar* определяется следующими синтаксическими правилами:

```
similar_predicate ::= source_value [NOT]
 SIMILAR TO pattern_value [ESCAPE escape_value]
```

```
source_value ::= character_expression
pattern_value ::= character_expression
escape_value ::= character_expression
```

Все три операнда ( `source_value`, `pattern_value` и `escape_value` ) должны иметь *тип символьных строк*. Значением последнего операнда должна быть строка из одного символа. Вторым операнд, как правило, задается литералом соответствующего типа. В обоих случаях значение *предиката* равняется `true` в том и только в том случае, когда шаблон ( `pattern_value` ) должным образом сопоставляется с исходной строкой ( `source_value` ).

Основное отличие предиката *similar* от рассмотренного ранее предиката *like* состоит в существенно расширенных возможностях задания шаблона, основанных на использовании правил построения **регулярных выражений**. *Регулярные выражения предиката similar* определяются следующими синтаксическими правилами:

```
regular_expression ::= regular_term
 | regular_expression vertical_bar regular_term
regular_term ::= regular_factor
 | regular_term regular_factor
regular_factor ::= regular_primary
 | regular_primary *
 | regular_primary +
regular_primary ::= character_specifier
 | %
 | regular_character_set
 | (regular_expression)
character_specifier ::= non_escape_character
 | escape_character
regular_character_set ::= _
 | left_bracket
 character_enumeration_list right_bracket
 | left_bracket
 ^ character_enumeration_list right_bracket
 | left_bracket : regular_charset_id : right_bracket
character_enumeration ::= character_specifier
 | character_specifier - character_specifier
regular_charset_id ::= ALPHA | UPPER | LOWER
 | DIGIT | ALNUM
```

Поскольку в синтаксических правилах *регулярных выражений* символы " | ", " [ " и " ] ", используемые нами в качестве *метасимволов* в *BNF*, являются *терминальными символами*, они изображены как `vertical_bar`, `left_bracket` и `right_bracket` соответственно.

Создаваемое по приведенным правилам регулярное выражение представляет собой символьную строку, содержащую все символы, которые требуется явно сопоставлять с символами строки-источника. В строке могут находиться специальные символы, представляющие собой заменители обычных символов ( " % " и " \_ " ), обозначения операций ( " | " ), показатели числа возможных повторений ( " \* " и " + " ) и т. д. При

вычислении регулярного выражения образуются все возможные *символьные строки*, не содержащие специальных символов и соответствующие исходному шаблону. Тем самым, значением предиката *similar* является **true** в том и только в том случае, когда среди всех *символьных строк*, генерируемых по регулярному выражению **pattern\_value**, найдется *символьная строка*, совпадающая с **source\_value**.

Рассмотрим несколько примеров *регулярных выражений*.

Выражение **'(This is string1)|(This is string2)'** производит две *символьные строки*: **'(This is string1)'** и **'(This is string2)'**. В общем случае в круглых скобках могут находиться произвольные *регулярные выражения rexp1 и rexp2*. Результатом вычисления **'(rexp1)|(rexp2)'** является множество *символьных строк*, генерируемых выражением **rexp1**, объединенное с множеством *символьных строк*, генерируемых выражением **rexp2**.

Выражение **'This is string [12]\*'** генерирует *символьные строки* **'This is string '**, **'This is string 1'**, **'This is string 2'**, **'This is string 11'**, **'This is string 22'**, **'This is string 12'**, **'This is string 21'**, **'This is string 111'** и т. д. Конструкция в квадратных скобках представляет собой один из вариантов определения *набора символов (regular\_character\_set)*. В данном случае символы, входящие в определяемый набор, просто перечисляются. При вычислении регулярного выражения в каждой из генерируемых *символьных строк* конструкция в квадратных скобках заменяется одним из символов соответствующего набора.

Специальный символ **"\*"**, стоящий после закрывающей квадратной скобки, является показателем числа повторений. "Звездочка" означает, что в генерируемых *символьных строках* элемент регулярного выражения, непосредственно предшествующий "звездочке", может появляться ноль или более раз. Использование в такой же ситуации специального символа **"+"** означает, что в генерируемых *символьных строках* элемент регулярного выражения, непосредственно предшествующий символу "плюс", может появляться один или более раз.

Другая форма определения *набора символов* иллюстрируется регулярным выражением **'This is string [:DIGIT:]'**. В этом случае конструкция в квадратных скобках представляет собой одиночный символ, изображающий десятичную цифру. Другими допустимыми в SQL *идентификаторах наборов символов (regular\_charset\_id)* являются **ALPHA** (любой символ алфавита), **UPPER** (любой символ верхнего регистра), **LOWER** (любой символ нижнего регистра) и **ALNUM** (любой алфавитно-цифровой символ).

Определяемый *набор символов* может задаваться нижней и верхней границей диапазона значений кодов допустимых символов. Например, в *регулярном выражении* **'This is string [3-8]'** конструкция в квадратных скобках представляет собой одиночный символ, изображающий цифры от 3 до 8 включительно. Заметим, что при задании диапазона можно использовать любые символы, но требуется, чтобы значение кода символа левой границы диапазона было не больше значения кода символа правой границы.

Наконец, имеется еще одна возможность определения *набора символов*. Соответствующая конструкция позволяет указать, какие символы из общего *набора символов SQL* не входят в определяемый *набор символов*. Например, регулярное выражение **'\_S[^\t]\*ing%'** генерирует все *символьные строки*, у которых вторым символом является **"S"**, за которым (не обязательно непосредственно) следует подстрока **"ing"**, но между **"S"** и **"ing"** отсутствуют вхождения символа **"t"**.

Как и в предикате *like*, символ, определенный в разделе *ESCAPE*, поставленный перед любым специальным символом, отменяет специальную интерпретацию этого символа.

В заключение данного пункта вернемся к отложенному в разделе "*Скалярные выражения*" лекции 13 обсуждению функции *SUBSTRING ... SIMILAR ... ESCAPE*. Напомним, что вызов этой функции определяется следующим синтаксисом:

```
SUBSTRING (character_value_expression
SIMILAR character_value_expression
ESCAPE character_value_expression)
```

Предположим, что в разделе *ESCAPE* (который должен присутствовать обязательно) задан символ "x". Тогда *символьная строка*, задаваемая во втором операнде, должна иметь вид *'rexp1x"rexp2x"rexp3'*, где *rexp1*, *rexp2* и *rexp3* являются регулярными выражениями. Функция пытается разделить символьную строку первого операнда на три раздела, первый из которых определяется путем сопоставления начала строки со строками, генерируемыми *rexp1*, второй - путем сопоставления оставшейся части строки первого операнда с *rexp2* и третий - путем сопоставления конца этой строки с *rexp3*. Возвращаемым значением функции является средняя часть символьной строки первого операнда.

Вот пример вызова функции:

```
SUBSTRING ('This is string22'
SIMILAR 'This is\'[:ALPHA:]+\'[:DIGIT:]+'
ESCAPE '\')
```

Результатом будет строка *'string'*.

#### Примеры запросов с использованием предиката *similar*

```
SELECT DEPT_NAME, DEPT_NO
FROM DEPT
WHERE DEPT_NAME SIMILAR TO
'(HARD|SOFT)WARE%\'[:DIGIT:]+\' ESCAPE \'';
```

14.14. Найти номера и названия отделов, название которых начинается со слов *'Hardware'* или *'Software'*, а за ними (не обязательно непосредственно) следует последовательность десятичных цифр, предваряемых символом подчеркивания.

```
SELECT DEPT_NAME, DEPT_NO
FROM DEPT
WHERE DEPT_NAME SIMILAR TO '[^1-9]+%';
```

14.15. Найти номера и названия проектов, название которых не начинается с последовательности цифр.

#### Предикат *exists*

Предикат *exists* определяется следующим синтаксическим правилом:

```
exists_predicate ::= EXISTS (query_expression)
```

Значением условия **EXISTS (query\_expression)** является **true** в том и только в том случае, когда мощность таблицы-результата выражения запросов больше нуля, иначе значением условия является **false**.

#### Примеры запросов с использованием предиката exists

```
SELECT DEPT.DEPT_NO
FROM DEPT
WHERE EXISTS
 (SELECT EMP.EMP_NO
 FROM EMP
 WHERE EMP.DEPT_NO = DEPT.DEPT_NO
 AND EXISTS
 (SELECT PRO.PRO_MNG
 FROM PRO
 WHERE PRO.PRO_MNG = EMP.EMP_NO));
```

14.16. Найти номера отделов, среди служащих которых имеются менеджеры проектов.

Эту формулировку можно упростить, избавившись от самого вложенного запроса ( [пример 14.16.1](#)):

```
SELECT DEPT.DEPT_NO
FROM DEPT
WHERE EXISTS
 (SELECT EMP.EMP_NO
 FROM EMP, PRO
 WHERE EMP.DEPT_NO = DEPT.DEPT_NO
 AND PRO.PRO_MNG = EMP.EMP_NO);
```

14.16.1.

Далее заметим, что по смыслу *предиката* **EXISTS** список выборки во вложенном *подзапросе* является несущественным, и формулировку запроса можно изменить, например, следующим образом ( [пример 14.16.2](#)):

```
SELECT DEPT.DEPT_NO
FROM DEPT
WHERE EXISTS
 (SELECT *
 FROM EMP, DEPT
 WHERE EMP.DEPT_NO = DEPT.DEPT_NO
 AND PRO.PRO_MNG = EMP.EMP_NO);
```

14.16.2.

Запросы с предикатом **EXISTS** можно также переформулировать в виде запросов с предикатом сравнения ( [пример 14.16.3](#)):

```
SELECT DEPT.DEPT_NO
FROM DEPT
WHERE (SELECT COUNT(*)
```

```
FROM EMP, DEPT
WHERE EMP.DEPT_NO = DEPT.DEPT_NO
AND PRO.PRO_MNG = EMP.EMP_NO) >= 1;
```

14.16.3.

```
SELECT DEPT.DEPT_NO
FROM DEPT
WHERE NOT EXISTS
(SELECT *
FROM EMP EMP1, EMP EMP2
WHERE EMP1.EMP_NO = DEPT.DEPT_MNG AND
EMP2.DEPT_NO = DEPT.DEPT_NO AND
EMP2.EMP_SAL > EMP1.EMP_SAL);
```

14.17. Найти номера отделов, размер заработной платы служащих которых не превышает размер заработной платы руководителя отдела.

### Предикат **unique**

Этот *предикат* позволяет сформулировать условие отсутствия дубликатов в результате запроса:

`unique_predicate ::= UNIQUE (query_expression)`

Результатом вычисления условия **UNIQUE (query\_expression)** является **true** в том и только в том случае, когда в таблице-результате выражения запросов отсутствуют какие-либо две строки, одна из которых является дубликатом другой. В противном случае значение условия есть **false**.

### Примеры запросов с использованием предиката **unique**

```
SELECT DEPT_NO
FROM DEPT
WHERE UNIQUE
(SELECT EMP_NAME, EMP_BDATE
FROM EMP
WHERE EMP.DEPT_NO = DEPT.DEPT_NO);
```

14.18. Найти номера отделов, служащих которых можно различить по имени и дате рождения.

Возможна альтернативная, но более сложная формулировка этого запроса с использованием предиката **NOT EXISTS** ( [пример 14.18.1](#)):

```
SELECT DEPT_NO
FROM DEPT
WHERE NOT EXISTS
(SELECT *
FROM EMP, EMP EMP1
WHERE EMP1.EMP_NO <> EMP.EMP_NO
AND EMP.DEPT_NO = DEPT.DEPT_NO
AND EMP1.DEPT_NO = DEPT.DEPT_NO
AND EMP1.EMP_NAME = EMP.EMP_NAME
```

```
AND(EMP1.EMP_BDATE = EMP.EMP_BDATE
OR (EMP.EMP_BDATE IS NULL
AND EMP1.EMP_BDATE IS NULL)));
```

14.18.1.

Если же ограничиться требованием уникальности имен служащих, то возможна следующая формулировка ( [пример 14.18.2](#)):

```
SELECT DEPT_NO
FROM DEPT
WHERE (SELECT COUNT (EMP_NAME)
FROM EMP
WHERE EMP.DEPT_NO = DEPT.DEPT_NO) =
(SELECT COUNT (DISTINCT EMP_NAME)
FROM EMP
WHERE EMP.DEPT_NO = DEPT.DEPT_NO);
```

14.18.2.

### Предикат overlaps

Этот *предикат* служит для проверки перекрытия во времени двух событий. Условие определяется следующим синтаксисом:

```
overlaps_predicate ::= row_value_constructor OVERLAPS
row_value_constructor
```

Степень каждой из строк-операндов должна быть равна 2. Тип данных первого столбца каждого из операндов должен быть типом даты-времени, и типы данных первых столбцов должны быть *совместимы*. Тип данных второго столбца каждого из операндов должен быть типом даты-времени или интервала. При этом:

- если это тип интервала, то точность типа должна быть такой, чтобы интервал можно было прибавить к значению *типа дата-время* первого столбца;
- если это *тип дата-время*, то он должен быть совместим с типом данных дата-время первого столбца.

Пусть **D1** и **D2** - значения первого столбца первого и второго операндов соответственно. Если второй столбец первого операнда имеет *тип дата-время*, то пусть **E1** обозначает его значение. Если второй столбец первого операнда имеет тип **INTERVAL**, то пусть **I1** -его значение, а **E1 = D1 + I1**. Если **D1** является неопределенным значением или если **E1 < D1**, то пусть **S1 = E1** и **T1 = D1**. В противном случае, пусть **S1 = D1** и **T1 = E1**. Аналогично определяются **S2** и **T2** применительно ко второму операнду. Результат условия совпадает с результатом вычисления следующего *булевского выражения*:

```
(S1 > S2 AND NOT (S1 >= T2 AND T1 >= T2))
OR
(S2 > S1 AND NOT (S2 >= T1 AND T2 >= T1))
OR
(S1 = S2 AND (T1 <> T2 OR T1 = T2))
```

**Примеры запросов с использованием предиката overlaps**

```
SELECT PRO_NO
FROM PRO
WHERE (PRO_SDATE, PRO_DURAT) OVERLAPS
 (DATE '2000-01-15', DATE '2002-12-31');
```

14.19. Найти номера проектов, которые выполнялись в период с 15 января 2000 г. по 31 декабря 2002 г.

```
SELECT PRO_TITLE
FROM PRO
WHERE (PRO_SDATE, PRO_DURAT) OVERLAPS
 (CURRENT_DATE, INTERVAL '1' YEAR);
```

14.20. Найти названия проектов, которые будут выполняться в течение следующего года.

#### Предикат сравнения с квантором

Этот *предикат* позволяет специфицировать квантифицированное сравнение строчного значения и определяется следующим синтаксическим правилом:

```
quantified_comparison_predicate ::= row_value_constructor
 comp_op { ALL | SOME | ANY } query_expression
```

Степень первого операнда должна быть такой же, как и степень таблицы-результата выражения запросов. Типы данных значений строки-операнда должны быть совместимы с типами данных соответствующих столбцов выражения запроса. Сравнение строк производится по тем же правилам, что и для *предиката сравнения*.

Обозначим через **x** строку-первый операнд, а через **S** - результат *вычисления* выражения запроса. Пусть **s** обозначает произвольную строку таблицы **S**. Тогда:

- условие **x comp\_op ALL S** имеет значение **true** в том и только в том случае, когда **S** пусто, или значение условия **x comp\_op s** равно **true** для каждой строки **s**, входящей в **S**. Условие **x comp\_op ALL S** имеет значение **false** в том и только в том случае, когда значение *предиката* **x comp\_op s** равно **false** хотя бы для одной строки **s**, входящей в **S**. В остальных случаях значение условия **x comp\_op ALL S** равно **unknown** ;
- условие **x comp\_op SOME S** имеет значение **false** в том и только в том случае, когда **S** пусто, или значение условия **x comp\_op s** равно **false** для каждой строки **s**, входящей в **S**. Условие **x comp\_op SOME S** имеет значение **true** в том и только в том случае, когда значение *предиката* **x comp\_op s** равно **true** хотя бы для одной строки **s**, входящей в **S**. В остальных случаях значение условия **x comp\_op SOME S** равно **unknown** ;
- условие **x comp\_op ANY S** эквивалентно условию **x comp\_op SOME S**.

#### Примеры запросов с использованием предиката сравнения с квантором

```
SELECT EMP_NO
FROM EMP
WHERE DEPT_NO = 65
 AND EMP_SAL > SOME (SELECT EMP1.EMP_SAL
 FROM EMP EMP1
 WHERE EMP.DEPT_NO = EMP1.DEPT_NO);
```

14.21. Найти номера служащих отдела номер 65, зарплата которых в этом отделе не является минимальной.

Одна из возможных альтернативных формулировок этого запроса может основываться на использовании предиката *EXISTS* ( [пример 14.21.1](#)):

```
SELECT EMP_NO
FROM EMP
WHERE DEPT_NO = 65
 AND EXISTS(SELECT *
 FROM EMP EMP1
 WHERE EMP.DEPT_NO = EMP1.DEPT_NO
 AND EMP.EMP_SAL > EMP1.EMP_SAL);
```

14.21.1.

Вот альтернативная формулировка этого запроса, основанная на использовании *агрегатной функции MIN* ( [пример 14.21.2](#)):

```
SELECT EMP_NO
FROM EMP
WHERE DEPT_NO = 65 AND
 EMP_SAL > (SELECT MIN(EMP1.EMP_SAL)
 FROM EMP EMP1
 WHERE EMP.DEPT_NO = EMP1.DEPT_NO);
```

14.21.2.

```
SELECT EMP_NO, EMP_NAME
FROM EMP
WHERE DEPT_NO = 65 AND
 EMP_NAME = SOME (SELECT EMP1.EMP_NAME
 FROM EMP EMP1
 WHERE EMP.DEPT_NO = EMP1.DEPT_NO
 AND EMP.EMP_NO <> EMP1.EMP_NO);
```

14.22. Найти номера и имена служащих отдела 65, однофамильцы которых работают в этом же отделе.

Заметим, что эта формулировка эквивалентна следующей формулировке ( [пример 14.22.1](#)):

```
SELECT EMP_NO, EMP_NAME
FROM EMP
WHERE DEPT_NO = 65 AND
 EMP_NAME IN (SELECT EMP1.EMP_NAME
 FROM EMP EMP1
 WHERE EMP.DEPT_NO = EMP1.DEPT_NO
 AND EMP.EMP_NO <> EMP1.EMP_NO);
```

14.22.1.

Возможна формулировка с использованием *агрегатной функции COUNT* ( [пример 14.22.2](#)):

```
SELECT EMP_NO, EMP_NAME
```

```

FROM EMP
WHERE DEPT_NO = 65 AND
 (SELECT COUNT(*)
 FROM EMP EMP1
 WHERE EMP.DEPT_NO = EMP1.DEPT_NO
 AND EMP.EMP_NO <> EMP1.EMP_NO) >= 1;

```

14.22.2.

Наиболее лаконичным образом этот запрос можно сформулировать с использованием соединения ( [пример 14.22.3](#)):

```

SELECT DISTINCT EMP.EMP_NO, EMP.EMP_NAME
FROM EMP, EMP EMP1
WHERE EMP.DEPT_NO = 65
 AND EMP.EMP_NAME = EMP1.EMP_NAME
 AND EMP.DEPT_NO = EMP1.DEPT_NO
 AND EMP.EMP_NO <> EMP1.EMP_NO;

```

14.22.3.

В последней формулировке мы вынуждены везде использовать уточненные имена столбцов, потому что на одном уровне используются два вхождения таблицы *EMP*.

```

SELECT EMP_NO
FROM EMP
WHERE DEPT_NO = 65
 AND EMP_SAL >= ALL(SELECT EMP1.EMP_SAL
 FROM EMP EMP1
 WHERE EMP.DEPT_NO = EMP1.DEPT_NO);

```

14.23. Найти номера служащих отдела номер 65, зарплата которых в этом отделе является максимальной.

Одна из возможных альтернативных формулировок этого запроса может основываться на использовании предиката **NOT EXISTS** ( [пример 14.23.1](#)):

```

SELECT EMP_NO
FROM EMP
WHERE DEPT_NO = 65
 AND NOT EXISTS (SELECT *
 FROM EMP EMP1
 WHERE EMP.DEPT_NO = EMP1.DEPT_NO
 AND EMP.EMP_SAL < EMP1.EMP_SAL);

```

14.23.1.

Можно сформулировать этот же запрос с использованием *агрегатной функции* **MAX** ( [пример 14.23.2](#)):

```

SELECT EMP_NO
FROM EMP

```

```
WHERE DEPT_NO = 65
AND EMP_SAL = (SELECT MAX(EMP1.EMP_SAL)
FROM EMP EMP1
WHERE EMP.DEPT_NO = EMP1.DEPT_NO);
```

14.23.2.

```
SELECT EMP_NO, EMP_NAME
FROM EMP
WHERE EMP_NAME <> ALL (SELECT EMP1.EMP_NAME
FROM EMP EMP1
WHERE EMP1.EMP_NO <> EMP.EMP_NO);
```

14.24. Найти номера и имена служащих, не имеющих однофамильцев

Этот запрос можно переформулировать на основе использования предиката **NOT EXISTS** или *агрегатной функции COUNT* (по причине очевидности мы не приводим эти формулировки), но, в отличие от случая в [примере 14.22.3](#), формулировка в виде запроса с соединением здесь не проходит. Формулировка запроса

```
SELECT DISTINCT EMP_NO, EMP_NAME
FROM EMP, EMP EMP1
WHERE EMP.EMP_NAME <> EMP1.EMP_NAME
AND EMP1.EMP_NO <> EMP.EMP_NO);
```

эквивалентна формулировке

```
SELECT EMP_NO, EMP_NAME
FROM EMP
WHERE EMP_NAME <> SOME (SELECT EMP1.EMP_NAME
FROM EMP EMP1
WHERE EMP1.EMP_NO <> EMP.EMP_NO);
```

Очевидно, что этот запрос является бессмысленным ("Найти служащих, для которых имеется хотя бы один не однофамилец").

## Предикат match

*Предикат* позволяет сформулировать условие соответствия строчного значения результату *табличного подзапроса*. Синтаксис определяется следующим правилом:

```
match_predicate ::= row_value_constructor
MATCH [UNIQUE] [SIMPLE | PARTIAL | FULL]
query_expression
```

Степень первого операнда должна совпадать со степенью таблицы-результата выражения запроса. *Типы данных столбцов* первого операнда должны быть совместимы с типами соответствующих столбцов *табличного подзапроса*. Сравнение пар соответствующих значений производится аналогично тому, как это специфицировалось для *предиката сравнения*.

Пусть **x** обозначает строку-первый операнд. Тогда:

- Если отсутствует спецификация вида сопоставления или специфицирован тип сопоставления *SIMPLE*, то:
  - если значение некоторого столбца *x* является неопределенным, то значением условия является *true* ;
  - если в *x* нет неопределенных значений, то:
    - если не указано *UNIQUE*, и в результате выражения запроса существует (возможно, не уникальная) строка *s* в такая, что  $x = s$ , то значением условия является *true* ;
    - если указано *UNIQUE*, и в результате выражения запроса существует уникальная строка *s*, такая, что  $x = s$ , то значением условия является *true* ;
    - в противном случае значением условия является *false*.
- Если в условии присутствует спецификация *PARTIAL*, то:
  - если все значения в *x* являются неопределенными, то значение условия есть *true* ;
  - иначе:
    - если не указано *UNIQUE*, и в результате выражения запроса существует (возможно, не уникальная) строка *s*, такая, что каждое отличное от неопределенного значение *x* равно соответствующему значению *s*, то значение условия есть *true* ;
    - если указано *UNIQUE*, и в результате выражения запроса существует уникальная строка *s*, такая, что каждое отличное от неопределенного значение *x* равно соответствующему значению *s*, то значение условия есть *true* ;
    - в противном случае значение условия есть *false*.
- Если в условии присутствует спецификация *FULL*, то:
  - если все значения в *x* неопределенные, то значение условия есть *true* ;
  - если ни одно значение в *x* не является неопределенным, то:
    - если не указано *UNIQUE*, и в результате выражения запроса существует (возможно, не уникальная) строка *s*, такая, что  $x = s$ , то значение условия есть *true* ;
    - если указано *UNIQUE*, и в результате выражения запроса существует уникальная строка *s*, такая, что  $x = s$ , то значение условия есть *true* ;
    - в противном случае значение условия есть *false*.
  - в противном случае значение условия есть *false*.

#### Примеры запросов с использованием предиката *match*

Все примеры этого пункта основаны на запросе "Найти номера служащих и номера их отделов для служащих, для которых в отделе со "схожим" номером работает служащий со "схожей" датой рождения" с некоторыми уточнениями.

```
SELECT EMP_NO, DEPT_NO
FROM EMP
WHERE (DEPT_NO, EMP_BDATE) MATCH SIMPLE
 (SELECT EMP1.DEPT_NO, EMP1.EMP_BDATE
 FROM EMP EMP1
 WHERE EMP1.EMP_NO <> EMP.EMP_NO);
14.25.
```

Этот запрос вернет данные о служащих, про которых:

- либо неизвестны номер отдела и дата рождения (или и то, и другое);
- либо в отделе данного служащего работает еще один человек с той же датой рождения.

Если использовать предикат *MATCH UNIQUE FULL*, то мы получим данные о служащих, про которых:

- либо неизвестны номер отдела **или** дата рождения (или и то, и другое);
- либо в отделе данного служащего работает еще один человек с той же датой рождения.

```
SELECT EMP_NO, DEPT_NO
FROM EMP
WHERE (DEPT_NO, EMP_BDATE) MATCH PARTIAL
 (SELECT EMP1.DEPT_NO, EMP1.EMP_BDATE
 FROM EMP EMP1
 WHERE EMP1.EMP_NO <> EMP.EMP_NO);
```

14.26.

Этот запрос вернет данные о служащих, про которых:

- либо неизвестны номер отдела **и** дата рождения;
- либо неизвестен номер отдела, но имеется по крайней мере еще один человек с той же датой рождения;
- либо неизвестна дата рождения, но в отделе данного служащего работает по крайней мере еще один человек;
- либо известны и номер отдела, и дата рождения, и в отделе данного служащего работает по крайней мере еще один человек с той же датой рождения.

Если использовать предикат *MATCH UNIQUE PARTIAL*, то мы получим данные о служащих, про которых:

- либо неизвестны номер отдела **и** дата рождения;
- либо неизвестен номер отдела, но имеется еще один человек с той же датой рождения;
- либо неизвестна дата рождения, но в отделе данного служащего работает еще один человек;
- либо известны и номер отдела, и дата рождения, и в отделе данного служащего работает еще один человек с той же датой рождения.

```
SELECT EMP_NO, DEPT_NO
FROM EMP
WHERE (DEPT_NO, EMP_BDATE) MATCH UNIQUE FULL
 (SELECT EMP1.DEPT_NO, EMP1.EMP_BDATE
 FROM EMP EMP1
 WHERE EMP1.EMP_NO <> EMP.EMP_NO);
```

14.27.

Этот запрос вернет данные о служащих, о которых:

- либо неизвестны номер отдела **и** дата рождения;
- либо в отделе данного служащего работает по крайней мере еще один человек с той же датой рождения.

Если использовать предикат *MATCH UNIQUE SIMPLE*, то мы получим данные о служащих, о которых:

- либо неизвестны номер отдела и дата рождения;
- либо в отделе данного служащего работает еще один человек с той же датой рождения.

### Предикат **is distinct**

*Предикат* позволяет проверить, являются ли две строки дубликатами. Условие определяется следующим синтаксическим правилом:

**distinct\_predicate ::= row\_value\_constructor IS DISTINCT FROM  
row\_value\_constructor**

Строки-операнды должны быть одинаковой степени. Типы данных соответствующих значений строк-операндов должны быть совместимы.

Напомним, что две строки **s1** с именами столбцов **c1, c2, ..., cn** и **s2** с именами столбцов **d1, d2, ..., dn** считаются строками-дубликатами, если для каждого **i** (**i = 1, 2, ..., n**) либо **ci** и **di** не содержат **NULL**, и **(ci = di) = true**, либо и **ci**, и **di** содержат **NULL**. Значением условия **s1 IS DISTINCT FROM s2** является **true** в том и только в том случае, когда строки **s1** и **s2** не являются дубликатами. В противном случае значением условия является **false**.

Заметим, что отрицательная форма условия - **IS NOT DISTINCT FROM** - в стандарте SQL не поддерживается. Вместо этого можно воспользоваться выражением **NOT s1 IS DISTINCT FROM s2**.

### Примеры запросов с использованием предиката **distinct**

```
SELECT EMP_NO, EMP_NAME
FROM EMP
WHERE DEPT_NO = 65
 AND (EMP_NAME, EMP_BDATE) IS DISTINCT FROM
 (SELECT EMP1.EMP_NAME, EMP1.EMP_BDATE
 FROM EMP EMP1, DEPT
 WHERE EMP1.DEPT_NO = EMP.DEPT_NO
 AND DEPT.DEPT_MNG = EMP1.EMP_NO);
```

14.28. Найти номера и имена служащих отдела 65, которых можно отличить по данным об имени и дате рождения от руководителя отдела 65.

```
SELECT EMP1.EMP_NO, EMP2.EMP_NO
FROM EMP EMP1, EMP EMP2
WHERE DEPT_NO = 65 AND EMP1.EMP_NO <> EMP2.EMP_NO
 AND NOT ((EMP1.EMP_NAME, EMP1.EMP_BDATE) IS DISTINCT FROM
 (EMP2.EMP_NAME, EMP2.EMP_BDATE));
```

14.29. Найти все пары номеров таких служащих отдела 65, которых нельзя различить по данным об имени и дате рождения.

### Заключение

В этой лекции мы обсудили наиболее важные возможности языка *SQL*, связанные с выборкой данных. Даже простые примеры, приводившиеся в лекции, показывают исключительную *избыточность* языка *SQL*. Еще в то время, когда действующим стандартом языка был *SQL/92*, была опубликована любопытная статья, в которой

приводилось 25 формулировок одного и того же несложного запроса. При использовании всех возможностей SQL:1999 этих формулировок было бы гораздо больше.

Можно спорить, хорошо или плохо иметь возможность формулировать один и тот же *запрос* десятками разных способов. На мой взгляд, это не очень хорошо, поскольку увеличивает *вероятность* появления ошибок в запросах (особенно в сложных запросах). С другой стороны, таково объективное состояние дел, и мы стремились обеспечить в этой лекции материал, достаточный для того, чтобы прочувствовать различные возможности формулировки запросов. Как показывают следующие две лекции, возможности, предоставляемые оператором **SELECT**, в действительности гораздо шире.

## Лабораторная работа 5. Группировка и условия раздела **HAVING**, порождаемые и соединенные таблицы

### Введение

В предыдущих двух лекциях мы обсудили допускаемые в стандарте SQL виды ссылок на таблицы в разделе **FROM** оператора **SELECT** и подробно, с многочисленными примерами, рассмотрели возможные способы построения условных выражений раздела **WHERE**. Данную лекцию мы начинаем с анализа возможностей и целесообразности использования в запросах разделов **GROUP BY** и **HAVING**. В ней обсуждаются виды *предикатов*, которые можно использовать в *условных выражениях* раздела **HAVING**, и приводятся иллюстрирующие примеры. Но в действительности мы преследуем большую цель: показать, что во многих случаях *разделы* **GROUP BY** и **HAVING** являются избыточными; *запрос* можно сформулировать более понятным образом без их использования. Применение разделов **GROUP BY** и **HAVING** оказывается действительно полезным, а иногда и необходимым, в тех случаях, когда в запросе присутствует несколько вызовов *агрегатных функций* на группах строк.

После обсуждения разделов **GROUP BY** и **HAVING** можно будет считать, что мы полностью рассмотрели базовые конструкции оператора выборки (раздел **ORDER BY** не заслуживает дополнительного обсуждения). Поэтому в разделах "Ссылки на порождаемые таблицы в разделе **FROM**" и "Более сложные конструкции оператора выборки" мы рассматриваем темы порождаемых таблиц, *соединенных таблиц* и *порождаемых таблиц с горизонтальной связью*.

В обычных порождаемых таблицах SQL нет ничего особенного. По всей видимости, возможность указывать в разделе **FROM** выражения запросов, а не только ссылки на базовые или представляемые таблицы, была введена в SQL на основе следующих естественных соображений. Результатом *вычисления выражения* запросов в SQL является *таблица*. Следовательно, в любой конструкции языка, где может присутствовать *ссылка* на таблицу SQL, следует допустить присутствие выражения запросов. Одновременное наличие возможностей определения представляемых таблиц, указания именованного выражения запросов в разделе **WITH** и указания выражения запросов порождаемой таблицы непосредственно в списке раздела **FROM**, очевидно, является избыточным.

*Соединенные таблицы* появились еще в стандарте SQL/92, и внедрение в стандарт SQL этой возможности было действительно обоснованным. В соответствии с традиционной общей семантикой оператора **SELECT** в нем вообще не предусматривались явные средства для выражения потребности в соединении двух или более таблиц. Наличие возможности указывать несколько ссылок на таблицы в разделе **FROM** и спецификации

произвольного логического выражения в разделе **WHERE** для ограничения расширенного декартова произведения этих таблиц позволяет выражать с помощью традиционных средств SQL соединение общего вида в смысле Кодда, и до поры до времени это считалось достаточным.

## Внешние соединения

Но имеются два важных частных случая соединений, которые выражаются с помощью традиционных средств SQL излишне громоздко,- это *естественные и внешние соединения*. При наличии возможности определения *внешних ключей* таблицы кажется достаточно странной потребность всякий раз явно указывать в запросах условие *естественного соединения*. Например, во многих примерах запросов в лекции 14 присутствует условие соединения **EMP.DEPT\_NO = DEPT.DEPT\_NO** в тех случаях, когда в действительности нам требовался результат операции **EMP NATURAL JOIN DEPT**.

Внешние соединения были введены еще Эдгаром Коддом в 1979 г. В целом, основная идея этой разновидности операции соединения состояла в том, что, с одной стороны, результат операции обычного соединения двух отношений повышает информационный уровень данных, поскольку в результате операции мы имеем информационно связанные данные. Но, с другой стороны, в результирующем отношении мы теряем информацию об исходных объектах, которые оказались несвязанными и не вошли в результат соединения. Кодд придумал, как, используя *неопределенные значения*, определить обобщенную операцию, которая будет обладать достоинствами обычной операции соединения, не приводя к потере исходной информации. Вернее, он предложил три операции: *левое внешнее соединение*, *правое внешнее соединение* и *полное (симметричное) внешнее соединение*. Приведем их определения (в реляционных терминах данного курса).

Пусть имеются отношения **r1** и **r2**, совместимые относительно операции взятия расширенного декартова произведения. Пусть **s** является результатом операции **r1 LEFT OUTER JOIN r2 WHERE comp** (левое внешнее соединение **r1** и **r2** по условию **comp**). Тогда  $H_s = H_{r1} \cup H_{r2}$ . Пусть  $tr1 \in Br1$  и  $tr2 \in Br2$ . Тогда  $tr1 \cup tr2 \in Bs$  в том и только в том случае, когда **comp** ( $tr1 \cup tr2$ ) = **true**. Если имеется кортеж  $tr1 \in Br1$ , для которого нет ни одного кортежа  $tr2 \in r2$ , такого, что **comp** ( $tr1 \cup tr2$ ) = **true**, то  $tr1 \cup tr2_{null} \in Bs$ , где  $tr2_{null}$  - кортеж, соответствующий **r2**, все значения которого являются неопределенными<sup>1</sup>.

Пусть **s** является результатом операции **r1 RIGHT OUTER JOIN r2 WHERE comp** (правое внешнее соединение **r1** и **r2** по условию **comp**). Тогда  $H_s = H_{r1} \cup H_{r2}$ . Пусть  $tr1 \in Br1$  и  $tr2 \in Br2$ . Тогда  $tr1 \cup tr2 \in Bs$  в том и только в том случае, когда **comp** ( $tr1 \cup tr2$ ) = **true**. Если имеется кортеж  $tr2 \in Br2$ , для которого нет ни одного такого кортежа  $tr1 \in Br1$ , что **comp** ( $tr1 \cup tr2$ ) = **true**, то  $tr1_{null} \cup tr2 \in Bs$ , где  $tr1_{null}$  - кортеж, соответствующий **r1**, все значения которого являются неопределенными.

Наконец, пусть **s** является результатом операции **r1 FULL OUTER JOIN r2 WHERE comp** (полное внешнее соединение **r1** и **r2** по условию **comp**). Тогда  $H_s = H_{r1} \cup H_{r2}$ . Пусть  $tr1 \in Br1$  и  $tr2 \in Br2$ . Тогда  $tr1 \cup tr2 \in Bs$

в том и только в том случае, когда  $\text{comp}(\text{tr1 union tr2}) = \text{true}$ . Если имеется кортеж  $\text{tr1} \in Br1$ , для которого нет ни одного кортежа  $\text{tr2} \in Br2$ , такого, что  $\text{comp}(\text{tr1 union tr2}) = \text{true}$ , то  $\text{tr1 union tr2}_{\text{null}} \in Bs$ , где  $\text{tr2}_{\text{null}}$  - кортеж, соответствующий  $Hr2$ , все значения которого являются неопределенными. Если имеется кортеж  $\text{tr2} \in Br2$ , для которого нет ни одного кортежа  $\text{tr1} \in Br1$ , такого, что  $\text{comp}(\text{tr1 union tr2}) = \text{true}$ , то  $\text{tr1}_{\text{null}} \text{ union tr2} \in Bs$ , где  $\text{tr1}_{\text{null}}$  - кортеж, соответствующий  $Hr1$ , все значения которого являются неопределенными.

Понятно, что традиционными средствами SQL можно выразить все виды внешних соединений (например, с использованием переключателей), но такие запросы будут очень громоздкими. Компании-производители SQL-ориентированных СУБД пытались обеспечивать выразительные средства внешних соединений путем расширения системы обозначений для операций сравнения. Этот подход был не слишком удачным и не обеспечивал общего решения.

В стандарте языка SQL специфицирован отдельный специализированный подязык для формирования выражений соединения таблиц. Такие выражения называются *соединенными таблицами*, и их можно использовать в качестве ссылок на таблицы в списке раздела **FROM**. Разработчики стандарта SQL не любят мельчить - в языке допускается 14 видов соединений:

- *прямое соединение* ;
- *внутреннее соединение по условию* ;
- *внутреннее соединение по совпадению значений указанных одноименных столбцов* ;
- *естественное внутреннее соединение* ;
- *левое внешнее соединение по условию* ;
- *правое внешнее соединение по условию* ;
- *полное внешнее соединение по условию* ;
- *левое внешнее соединение по совпадению значений указанных одноименных столбцов* ;
- *правое внешнее соединение по совпадению значений указанных одноименных столбцов* ;
- *полное внешнее соединение по совпадению значений указанных одноименных столбцов* ;
- *естественное левое внешнее соединение* ;
- *естественное правое внешнее соединение* ;
- *естественное полное внешнее соединение* ;
- *соединение объединением*.

Во всех этих операциях нет ничего сложного, но их неформальное описание исключительно громоздко. Поэтому в разделе "Более сложные конструкции оператора выборки" мы определяем операции на формальном уровне, а потом иллюстрируем их на примерах.

Наконец, последняя тема этой лекции относится к еще одному типу ссылок на таблицу, допускаемых в разделе **FROM**: *порождаемым таблицам с горизонтальной связью*. Фактически *порождаемая таблица с горизонтальной связью* представляет собой выражение запросов, в котором может присутствовать корреляция со строками таблиц, специфицированных в списке раздела **FROM** слева от данной *порождаемой таблицы с горизонтальной связью*. Наличие *порождаемых таблиц с горизонтальной связью* требует некоторого уточнения семантики выполнения раздела **FROM** оператора **SELECT**. По нашему мнению, это средство является полностью избыточным, хотя и не вредным,

поскольку его реализация не должна вызывать затруднений и/или снижать эффективность системы.

## Агрегатные функции, группировка и условия раздела HAVING

В этом разделе мы *систематически* обсудим все аспекты группировки таблиц и вычисления *агрегатных функций*. Некоторые темы уже затрагивались на неформальном уровне в предыдущих лекциях.

### Семантика агрегатных функций

*Агрегатные функции* (в стандарте SQL они называются функциями над множествами)<sup>2</sup> определяются следующими синтаксическими правилами:

```
<set_function_specification> ::=
 COUNT(*)
 | set_function_type ([DISTINCT | ALL] value_expression)
 | GROUPING (column_reference)
<set_function_type> ::=
 { AVG | MAX | MIN | SUM | EVERY | ANY | SOME | COUNT }
```

Как видно из этих правил, в *стандарте SQL:1999* определены пять стандартных *агрегатных функций*: **COUNT** - число строк или значений, **MAX** - максимальное значение, **MIN** - минимальное значение, **SUM** - суммарное значение и **AVG** - *среднее значение*, а также две "кванторные" функции **EVERY** и **SOME ( ANY )**. В последних двух случаях выражение должно иметь *булевский тип*. Обсуждение функции **GROUPING** мы отложим до следующей лекции.

*Агрегатные функции* предназначены для того, чтобы вычислять некоторое значение для заданного *мультимножества* строк. Таким *мультимножеством* строк может быть группа строк, если *агрегатная функция* применяется к сгруппированной таблице, или (в вырожденных случаях) вся таблица. Для всех *агрегатных функций*, кроме **COUNT(\*)**, фактический (т. е. требуемый семантикой) порядок вычислений состоит в следующем. На основании параметров *агрегатной функции* из заданного *мультимножества* строк производится список значений. Затем по этому *списку значений* производится вычисление функции. Если список оказался пустым, то значением функции **COUNT** для него является 0, значением функции **SOME** - **false**, значением функции **ALL** - **true**, а значением всех остальных функций - **NULL**.

Пусть **T** обозначает тип значений из этого списка (вернее, "наименьший общий" тип, см. раздел "*Скалярные выражения*" лекции 13). *Типы значений агрегатных функций* определяются следующими правилами.

- Результат вычисления функции **COUNT** - это точное число с точностью и шкалой, которые определяются в реализации.
- Тип результата значений функций **MAX** и **MIN** совпадает с **T**. При вычислении функций **SUM** и **AVG** тип **T** не должен быть *типом символьных строк*.
  - Если **T** представляет собой тип точных чисел, то и типом результата функции является тип точных чисел с определяемыми в реализации точностью и шкалой.
  - Если **T** представляет собой тип приближительных чисел, то и типом результата функции является тип приближительных чисел с определяемой в реализации точностью.

- Для функций **EVERY** и **SOME T** является булевским типом<sup>3</sup>.
  - Первая функция принимает значение **true** в том и только в том случае, когда *вычисление выражения-аргумента* дает значение **true** для каждой строки из заданного набора строк, и **false**, когда *значение выражения-аргумента* есть **false** хотя бы для одной строки из заданного набора строк.
  - Функция **SOME** принимает значение **false** в том и только в том случае, когда *значение выражения-аргумента* есть **false** для каждой строки из заданного набора строк, и **true**, когда *значение выражения-аргумента* есть **true** хотя бы для одной строки из заданного набора строк.

Вычисление функции **COUNT(\*)** производится путем подсчета числа строк в заданном мультимножестве. Все строки считаются различными, даже если они состоят из одного столбца со значением **null** во всех строках.<sup>4</sup>

Если "арифметическая" (**AVG, MAX, MIN, SUM, COUNT**) агрегатная функция специфицирована с ключевым словом **DISTINCT**, то множество значений, на котором она вычисляется, строится из значений указанного выражения, вычисляемого для каждой строки заданной группы строк. Затем из этого мультимножества удаляются неопределенные значения, и в нем устраняются значения-дубликаты (т. е. образуется множество). После этого вычисляется указанная функция.

Если агрегатная функция специфицирована без ключевого слова **DISTINCT** (или с ключевым словом **ALL**), то мультимножество значений формируется из значений выражения, вычисляемого для каждой строки заданной группы строк. Затем из этого мультимножества удаляются неопределенные значения, и производится вычисление агрегатной функции.

## Результаты запросов и агрегатные функции

Об использовании агрегатных функций в разделах **HAVING** и **SELECT** оператора выборки упоминалось в разделе "Общие синтаксические правила построения скалярных выражений" лекции 13. В данном подразделе уместно повторить и уточнить этот материал.

Агрегатные функции можно разумным образом использовать в списке выборки (при построении выражений, являющихся элементами выборки) и в логическом выражении раздела **HAVING** (вернее, в выражениях, входящих в простые условия). Рассмотрим разные случаи применения агрегатных функций в списке выборки в зависимости от вида табличного выражения.

Если результат табличного выражения **R** не является сгруппированной таблицей (т. е. в табличном выражении отсутствуют разделы **GROUP BY** и **HAVING**), то появление в списке выборки хотя бы одного вызова агрегатной функции от (мульти) множества строк **R** приводит к тому, что **R** неявно рассматривается как сгруппированная таблица, состоящая из одной (или нуля, если **R** пусто) групп с отсутствующими столбцами группирования. Поэтому в данном случае в выражениях списка выборки не допускается прямое использование имен столбцов **R**: все они должны находиться внутри спецификаций вызова агрегатных функций. Результатом запроса является таблица, состоящая не более чем из одной строки, значения столбцов которой получены путем применения агрегатных функций к **R**.

Аналогично обстоит дело в том случае, когда **R** представляет собой сгруппированную таблицу, но табличное выражение не содержит раздела **GROUP BY** (и, следовательно, содержит раздел **HAVING**). В этом случае считается, что результат *табличного выражения* явно объявлен сгруппированной таблицей, состоящей из одной группы, и результат запроса можно формировать только путем применения *агрегатных функций* к данной группе строк. Опять результатом запроса является таблица, состоящая не более чем из одной строки, значения столбцов которой получены путем применения *агрегатных функций* к **R**.

Наконец, рассмотрим случай, когда **R** представляет собой "настоящую" сгруппированную таблицу, т. е. табличное выражение содержит раздел **GROUP BY**, и, следовательно, определен по крайней мере один столбец группирования (т. е. имеется хотя бы один такой столбец, что для любой группы его значения одинаковы во всех строках группы). В этом случае правила формирования списка выборки полностью соответствуют правилам формирования условия выборки раздела **HAVING**. Другими словами, в выражениях, являющихся элементами списка выборки, допускается прямое использование имен столбцов группирования, а спецификации остальных столбцов **R** могут появляться только внутри спецификаций *агрегатных функций*. Результатом запроса является таблица, число строк в которой равно числу групп в **R**. Значения столбцов каждой строки формируются на основе значений столбцов группирования и вызовов *агрегатных функций* для соответствующей группы.

### Логические выражения раздела HAVING

Приведем примеры использования в логических выражениях раздела **HAVING** некоторых предикатов, обсуждавшихся в предыдущей лекции. Теоретически в этих логических выражениях можно использовать всепредикаты, но применение тех предикатов, которые мы проиллюстрируем, является более естественным.

#### Предикаты сравнения

```
SELECT DEPT_NO
FROM EMP
WHERE DEPT_NO IS NOT NULL
GROUP BY DEPT_NO
HAVING COUNT(*) = 30;
```

15.1. Найти номера отделов, в которых работает ровно 30 служащих.

Конечно, этот запрос можно сформулировать и без использования разделов **GROUP BY** и **HAVING**. Например, возможна следующая формулировка ( [пример 15.1.1](#)):

```
SELECT DISTINCT DEPT_NO
FROM EMP
WHERE (SELECT COUNT (*)
 FROM EMP EMP1
 WHERE EMP1.DEPT_NO = EMP.DEPT_NO) = 30;
```

15.1.1.

Обратите внимание, что в формулировке [15.1.1](#) отдельная проверка условия **DEPT\_NO IS NOT NULL** не требуется.

```
SELECT DEPT_NO
FROM EMP
WHERE DEPT_NO IS NOT NULL
GROUP BY DEPT_NO
HAVING AVG(EMP_SAL) > 12000.00;
```

15.2. Найти номера всех отделов, в которых средний размер зарплаты служащих превосходит 12000 руб.

Очевидно, что и в этом случае возможна формулировка запроса без использования разделов **GROUP BY** и **HAVING** ( пример 15.2.1):

```
SELECT DISTINCT DEPT_NO
FROM EMP
WHERE (SELECT AVG(EMP1.EMP_SAL)
 FROM EMP EMP1
 WHERE EMP1.DEPT_NO = EMP.DEPT_NO) > 12000.00;
```

15.2.1.

Немного задержимся на этих примерах и обсудим, что означает различие в формулировках запросов. В соответствии с семантикой оператора **SELECT**, при выполнении запросов 15.1.1 и 15.2.1 для каждой строки таблицы **EMP** в цикле просмотра внешнего запроса будет выполняться *подзапрос*, который в случае наших примеров выберет из таблицы **EMP** (**EMP1**) все строки со значением столбца **DEPT\_NO**, равным значению этого столбца в текущей строке внешнего цикла. Другими словами, для каждой строки внешнего цикла образуется группа, для нее проверяется условие выборки, и в списке выборки используется имя столбца этой неявной группировки. Из-за того, что группа образуется и оценивается для каждой строки таблицы **EMP**, мы вынуждены указать в разделе **SELECT** спецификацию **DISTINCT**.

Формулировки 15.1 и 15.2 обеспечивают более четкие указания для выполнения запроса. Нужно сразу сгруппировать таблицу **EMP** в соответствии со значениями столбца **DEPT\_NO**, отобрать нужные группы, и для каждой отобранной группы вычислить значения выражений списка выборки. В этом случае семантика выполнения запроса не предписывает выполнения лишних действий. Конечно, в развитой реализации SQL компилятор должен суметь понять, что формулировки 15.1.1 и 15.2.1 эквивалентны формулировкам 15.1 и 15.2 соответственно, и избежать выполнения лишних действий.

```
SELECT DEPT_NO
FROM EMP
WHERE DEPT_NO IS NOT NULL
GROUP BY DEPT_NO
HAVING SUM(EMP_SAL) < (SELECT SUM(EMP1.EMP_SAL)
 FROM EMP EMP1, DEPT
 WHERE EMP1.EMP_NO = DEPT_MNG);
```

15.3. Найти номера всех отделов, в которых суммарный объем зарплаты служащих меньше суммарного объема зарплаты всех руководителей отделов.

И в этом случае возможна формулировка без использования разделов **GROUP BY** и **HAVING** ( [пример 15.3.1](#)). Эта формулировка является более сложной, чем в случае двух предыдущих примеров, но и к ней применимы приведенные выше замечания.

```
SELECT DISTINCT DEPT_NO
FROM EMP
WHERE (SELECT SUM(EMP1.EMP_SAL)
 FROM EMP EMP1
 WHERE EMP1.DEPT_NO = EMP.DEPT_NO) <
 (SELECT SUM(EMP1.EMP_SAL)
 FROM EMP EMP1, DEPT
 WHERE EMP1.EMP_NO = DEPT_MNG);
```

15.3.1.

```
SELECT DEPT.DEPT_NO, EMP.EMP_NAME, COUNT(*),
 MIN(EMP1.EMP_SAL), MAX(EMP1.EMP_SAL), AVG(EMP1.EMP_SAL)
FROM DEPT, EMP, EMP EMP1
WHERE DEPT.DEPT_NO = EMP1.DEPT_NO
GROUP BY DEPT.DEPT_NO, DEPT.DEPT_MNG, EMP.EMP_NO, EMP.EMP_NAME
HAVING DEPT.DEPT_MNG = EMP.EMP_NO;
```

15.4. Для каждого отдела найти его номер, имя руководителя, число служащих, минимальный, максимальный и средний размеры зарплаты служащих.

Этот запрос иллюстрирует несколько интересных особенностей языка SQL. Во-первых, это первый пример запроса с соединениями, в котором присутствуют разделы **GROUP BY** и **HAVING**. Во-вторых, одно условие соединения находится в разделе **WHERE**, а другое - в разделе **HAVING**. На самом деле, можно было бы перенести в раздел **WHERE** и второе условие соединения, и, скорее всего, на практике использовалась бы формулировка, приведенная в [примере 15.4.1](#):

```
SELECT DEPT.DEPT_NO, EMP.EMP_NAME, COUNT(*),
 MIN(EMP1.EMP_SAL), MAX(EMP1.EMP_SAL), AVG(EMP1.EMP_SAL)
FROM DEPT, EMP, EMP EMP1
WHERE DEPT.DEPT_NO = EMP1.DEPT_NO
 AND DEPT.DEPT_MNG = EMP.EMP_NO
GROUP BY DEPT.DEPT_NO, EMP.EMP_NAME;
```

15.4.1.

Но первая формулировка тоже верна, поскольку второе условие соединения определено на столбцах группировки.

Наконец, легко видеть, что, по существу, группировка производится по значениям столбца **DEPT.DEPT\_NO**. Остальные столбцы, указанные в списке столбцов группировки, функционально определяются столбцом **DEPT.DEPT\_NO**. Тем не менее, в первой формулировке мы включили в этот список столбцы **DEPT.DEPT\_MNG** и **EMP.EMP\_NO**, чтобы их имена можно было использовать в условии раздела **HAVING**, и столбец **EMP.EMP\_NAME**, чтобы можно было использовать его имя в списке выборки раздела **SELECT**. Другими словами, мы вынуждены расширять запрос избыточными данными, чтобы выполнить формальные синтаксические требования языка. Как видно, во второй формулировке мы смогли удалить из списка группировки два столбца. Кстати, не

следует думать, что многословие первой формулировки мешает СУБД выполнить запрос настолько же эффективно, как запрос во второй формулировке. Грамотно построенный оптимизатор SQL сам приведет первую формулировку ко второй.

Наконец, и этот запрос можно сформулировать без использования раздела **GROUP BY** за счет использования *подзапросов* в списке раздела **SELECT** ( [пример 15.4.2](#)):

```
SELECT DEPT.DEPT_NO, EMP.EMP_NAME,
 (SELECT COUNT(*)
 FROM EMP
 WHERE EMP.DEPT_NO = DEPT.DEPT_NO),
 (SELECT MIN(EMP_SAL)
 FROM EMP
 WHERE EMP.DEPT_NO = DEPT.DEPT_NO),
 (SELECT MAX(EMP_SAL)
 FROM EMP
 WHERE EMP.DEPT_NO = DEPT.DEPT_NO),
 (SELECT AVG(EMP_SAL)
 FROM EMP
 WHERE EMP.DEPT_NO = DEPT.DEPT_NO)
FROM DEPT, EMP
WHERE DEPT.DEPT_MNG = EMP.EMP_NO;
15.4.2.
```

Здесь мы снова имеем замаскированную группировку строк по значениям столбца **DEPT.DEPT\_NO** и вычисление *агрегатных функций* для каждой группы. Формально группа строится каждый раз заново при вызове каждой *агрегатной функции*. Хороший компилятор SQL должен привести формулировку [15.4.2](#) к виду [15.4.1](#).

И последнее замечание. Во всех приведенных формулировках в результат не попадут данные об отделах, в которых отсутствует руководитель (столбец **DEPT.DEPT\_MNG** может содержать *неопределенное значение*). Вообще говоря, это не противоречит условию запроса, но если бы мы хотели выдавать в результате **NULL** в качестве имени руководителя отдела с отсутствующим руководителем, то можно было немного усложнить формулировку запроса, например, следующим образом ( [пример 15.4.3](#)):

```
SELECT DEPT.DEPT_NO,
 CASE WHEN DEPT.DEPT_MNG IS NULL THEN NULL
 ELSE (SELECT EMP.EMP_NAME
 FROM EMP
 WHERE EMP.EMP_NO = DEPT.DEPT_MNG),
 COUNT(*), MIN(EMP1.EMP_SAL),
 MAX(EMP1.EMP_SAL), AVG(EMP1.EMP_SAL)
FROM DEPT, EMP, EMP EMP1
WHERE DEPT.DEPT_NO = EMP1.DEPT_NO
GROUP BY DEPT.DEPT_NO;
15.4.3.
```

**Предикат between**

```

SELECT DEPT_NO, MIN(EMP_SAL), MAX(EMP_SAL)
FROM EMP
WHERE DEPT_NO IS NOT NULL
GROUP BY DEPT_NO
HAVING AVG(EMP_SAL) BETWEEN
 (SELECT AVG(EMP_SAL)
 FROM EMP) AND 30000.00;

```

15.5. Найти номера отделов и минимальный и максимальный размер зарплаты служащих для отделов, в которых средний размер зарплаты служащих не меньше среднего размера зарплаты служащих во всей компании и не больше 30000 руб.

Еще раз приведем возможную формулировку этого запроса без использования разделов **GROUP BY** и **HAVING** ( [пример 15.5.1](#)):

```

SELECT DISTINCT DEPT_NO, (SELECT MIN(EMP1.EMP_SAL)
 FROM EMP EMP1
 WHERE EMP1.DEPT_NO = EMP.DEPT_NO),
 (SELECT MAX(EMP1.EMP_SAL)
 FROM EMP EMP1
 WHERE EMP1.DEPT_NO = EMP.DEPT_NO)
FROM EMP
WHERE (SELECT AVG(EMP1.EMP_SAL)
 FROM EMP EMP1
 WHERE EMP1.DEPT_NO = EMP.DEPT_NO) BETWEEN
 (SELECT AVG(EMP_SAL)
 FROM EMP) AND 30000.00;

```

#### 15.5.1.

Как видно, отказ от использования раздела **GROUP BY** приводит к размножению однотипных *подзапросов*, строящих одну и ту же группу строк, над которой вычисляется *агрегатная функция*.

#### Предикат null

```

SELECT DEPT.DEPT_NO, COUNT(*)
FROM DEPT, EMP EMP1, EMP EMP2
WHERE DEPT.DEPT_NO = EMP2.DEPT_NO
 AND DEPT.DEPT_MNG = EMP1.EMP_NO
GROUP BY DEPT.DEPT_NO, EMP1.DEPT_NO
HAVING EMP1.DEPT_NO IS NULL;

```

15.6. Найти номера и число служащих отделов, данные о руководителях которых не содержат номер отдела (конечно, в этом случае нас интересуют только те отделы, у которых имеется руководитель).

Как и в [примере 15.4](#), условие раздела **HAVING** можно переместить в раздел **WHERE** и получить вторую формулировку ( [пример 15.6.1](#)):

```

SELECT DEPT.DEPT_NO, COUNT(*)
FROM DEPT, EMP EMP1, EMP EMP2

```

```

WHERE DEPT.DEPT_NO = EMP2.DEPT_NO AND
 DEPT.DEPT_MNG = EMP1.EMP_NO AND
 EMP1.DEPT_NO IS NULL
GROUP BY DEPT.DEPT_NO;
15.6.1.

```

Кстати, в этом случае, поскольку в запросе присутствует только один вызов *агрегатной функции*, формулировка без использования раздела **GROUP BY** оказывается более понятной и не менее эффективной (даже при следовании предписанной семантике выполнения оператора **SELECT** ), что показывает [пример 15.6.2](#):

```

SELECT DEPT.DEPT_NO, (SELECT COUNT(*)
 FROM EMP
 WHERE DEPT.DEPT_NO = EMP.DEPT_NO)
FROM DEPT, EMP
WHERE DEPT.DEPT_MNG = EMP.EMP_NO AND
 EMP.DEPT_NO IS NULL;
15.6.2.

```

#### **Предикат in**

```

SELECT DEPT.DEPT_NO
FROM DEPT, EMP
WHERE DEPT.DEPT_NO = EMP.DEPT_NO
GROUP BY DEPT.DEPT_NO
HAVING AVG(EMP.EMP_SAL) IN
 (SELECT MAX(EMP1.EMP_SAL)
 FROM EMP, DEPT DEPT1
 WHERE EMP.DEPT_NO = DEPT1.DEPT_NO
 AND DEPT1.DEPT_NO <> DEPT.DEPT_NO
 GROUP BY DEPT.DEPT_NO);

```

15.7. Найти номера отделов, в которых средний размер зарплаты служащих равен максимальному размеру зарплаты служащих какого-либо другого отдела.

Этот запрос, помимо прочего, демонстрирует наличие в условии раздела **HAVING** вложенного *подзапроса* с корреляцией. Как и раньше, можно избавиться от разделов **GROUP BY** и **HAVING** во внешнем запросе ([пример 15.7.1](#)):

```

SELECT DEPT.DEPT_NO
FROM DEPT
WHERE (SELECT AVG(EMP_SAL)
 FROM EMP
 WHERE EMP.DEPT_NO = DEPT.DEPT_NO) IN
 (SELECT MAX(EMP1.EMP_SAL)
 FROM EMP, DEPT DEPT1
 WHERE EMP.DEPT_NO = DEPT1.DEPT_NO
 AND DEPT1.DEPT_NO <> DEPT.DEPT_NO
 GROUP BY DEPT.DEPT_NO);
15.7.1.

```

Но в данном случае мы не можем отказаться от раздела **GROUP BY** во втором вложенном запросе, поскольку без этого невозможно получить множество значений результатов вызова *агрегатной функции*.

#### **Предикат like**

```
SELECT EMP_NAME, COUNT(*)
FROM EMP, DEPT
WHERE EMP.DEPT_NO = DEPT.DEPT_NO
GROUP BY DEPT.DEPT_NO, EMP_NAME
HAVING COUNT(*) > 1
 AND EMP.EMP_NAME LIKE (SELECT EMP1.EMP_NAME
 FROM EMP EMP1
 WHERE EMP1.EMP_NO = DEPT.DEPT_MNG) || '%';
```

15.8. Во всех отделах найти имена и число служащих, у которых в данном отделе имеются однофамильцы и фамилии которых начинаются со строки символов, изображающей фамилию руководителя отдела.

Конечно, и в этом случае условие с *предикатом LIKE* можно переместить из раздела **HAVING** в раздел **WHERE**. Этот запрос можно переформулировать в виде, лишенном разделов **GROUP BY** и **HAVING** ([пример 15.8.1](#)), но вряд ли это разумно, поскольку формулировка является менее понятной и существенно более сложной.

```
SELECT EMP_NAME, (SELECT COUNT(*)
 FROM EMP EMP1
 WHERE EMP1.DEPT_NO = EMP.DEPT_NO
 AND EMP1.EMP_NAME = EMP.EMP_NAME
 AND EMP1.EMP_NO <> EMP.EMP_NO) + 1
FROM EMP
WHERE (SELECT COUNT(*)
 FROM EMP EMP1
 WHERE EMP1.DEPT_NO = EMP.DEPT_NO
 AND EMP1.EMP_NAME = EMP.EMP_NAME
 AND EMP1.EMP_NO <> EMP.EMP_NO) > 1
AND EMP_NAME LIKE (SELECT EMP1.EMP_NAME
 FROM EMP EMP1, DEPT
 WHERE EMP.DEPT_NO = DEPT.DEPT_NO
 AND EMP1.EMP_NO = DEPT.DEPT_MNG) || '%';
```

15.8.1.

#### **Предикат exists**

```
SELECT DEPT.DEPT_NO
FROM DEPT, EMP
WHERE DEPT.DEPT_NO = EMP.DEPT_NO
GROUP BY DEPT.DEPT_NO
HAVING EXISTS (SELECT *
 FROM EMP EMP1
 WHERE EMP1.DEPT_NO <> DEPT.DEPT_NO
 GROUP BY EMP1.DEPT_NO
 HAVING MAX (EMP1.EMP_SAL)= AVG (EMP.EMP_SAL));
```

15.9. Найти номера отделов, в которых средний размер зарплаты служащих равен максимальному размеру зарплаты служащих какого-либо другого отдела (другая формулировка для примера 15.7).

В этой формулировке основной интерес представляет *подзапрос*, в котором корреляция с внешним запросом происходит через вызов *агрегатной функции* от группы строк внешнего запроса. Здесь также можно избавиться от разделов **GROUP BY** и **HAVING** во внешнем запросе ( [пример 15.9.1](#)):

```
SELECT DEPT.DEPT_NO
FROM DEPT
WHERE EXISTS (SELECT EMP.DEPT_NO
 FROM EMP
 WHERE EMP.DEPT_NO <> DEPT.DEPT_NO
 GROUP BY EMP.DEPT_NO
 HAVING MAX (EMP.EMP_SAL)=
 (SELECT AVG (EMP1.EMP_SAL)
 FROM EMP EMP1
 WHERE EMP1.DEPT_NO = DEPT.DEPT_NO));
```

15.9.1.

**Предикат unique**

```
SELECT DEPT.DEPT_NO, AVG (EMP.EMP_SAL)
FROM DEPT, EMP
WHERE DEPT.DEPT_NO = EMP.DEPT_NO
GROUP BY DEPT.DEPT_NO
HAVING UNIQUE (SELECT AVG (EMP1.EMP_SAL)
 FROM EMP EMP1
 WHERE EMP1.DEPT_NO IS NOT NULL
 GROUP BY EMP1.DEPT_NO
 HAVING AVG (EMP1.EMP_SAL) = AVG (EMP.EMP_SAL));
```

15.10. Найти номера отделов и средний размер зарплаты служащих для таких отделов, где средний размер зарплаты служащих отличается от среднего размера зарплаты всех других отделов.

Вот альтернативная формулировка этого запроса с использованием *предиката NOT EXISTS* ( [пример 15.10.1](#)):

```
SELECT DEPT.DEPT_NO, AVG (EMP.EMP_SAL)
FROM DEPT, EMP
WHERE DEPT.DEPT_NO = EMP.DEPT_NO
GROUP BY DEPT.DEPT_NO
HAVING NOT EXISTS (SELECT EMP1.DEPT_NO
 FROM EMP EMP1
 WHERE EMP1.DEPT_NO <> DEPT.DEPT_NO
 GROUP BY EMP1.DEPT_NO
 HAVING AVG (EMP1.EMP_SAL)= AVG (EMP.EMP_SAL));
```

15.10.1.

**Предикаты сравнения с квантором**

```

SELECT DEPT_NO, AVG (CURRENT_DATE - EMP_BDATE)
FROM EMP
WHERE DEPT_NO IS NOT NULL
GROUP BY DEPT_NO
HAVING AVG (CURRENT_DATE - EMP_BDATE) < SOME
 (SELECT AVG (CURRENT_DATE - EMP1.EMP_BDATE)
 FROM EMP EMP1
 WHERE EMP1.DEPT_NO IS NOT NULL
 GROUP BY EMP1.DEPT_NO);

```

15.11. Найти номера отделов и средний возраст служащих для таких отделов, что найдется хотя бы один другой отдел, средний возраст служащих которого больше, чем в данном.

Напомним, что "ниладическая" функция **CURRENT\_DATE** выдает текущую дату, и, следовательно, *значением выражения* **CURRENT\_DATE - EMP\_BDATE** является интервал, представляющий текущий возраст служащего. На наш взгляд, формулировка этого запроса несколько упрощается, если пользоваться *предикатом* **EXISTS** ( *пример 15.11.1*):

```

SELECT DEPT_NO, AVG (CURRENT_DATE - EMP_BDATE)
FROM EMP
WHERE DEPT_NO IS NOT NULL
GROUP BY DEPT_NO
HAVING EXISTS (SELECT EMP1.DEPT_NO
 FROM EMP EMP1
 WHERE EMP1.DEPT_NO IS NOT NULL
 GROUP BY EMP1.DEPT_NO
 HAVING AVG (CURRENT_DATE - EMP1.EMP_BDATE) >
 AVG (CURRENT_DATE - EMP.EMP_BDATE));

```

15.11.1.

```

SELECT DEPT_NO, AVG (CURRENT_DATE - EMP_BDATE)
FROM EMP
WHERE DEPT_NO IS NOT NULL
GROUP BY DEPT_NO
HAVING AVG (CURRENT_DATE - EMP_BDATE) <= ALL
 (SELECT AVG (CURRENT_DATE - EMP_BDATE)
 FROM EMP
 WHERE DEPT_NO IS NOT NULL
 GROUP BY DEPT_NO);

```

15.12. Найти номера отделов и средний возраст служащих для отделов с минимальным средним возрастом служащих.

Этот запрос легко формулируется в более понятном виде с использованием *предиката* **NOT EXISTS** ( *пример 15.12.1*):

```

SELECT DEPT_NO, AVG (CURRENT_DATE - EMP_BDATE)
FROM EMP
WHERE DEPT_NO IS NOT NULL
GROUP BY DEPT_NO

```

```
HAVING NOT EXISTS (SELECT EMP1.DEPT_NO
FROM EMP EMP1
WHERE EMP1.DEPT_NO IS NOT NULL
GROUP BY EMP1.DEPT_NO
HAVING AVG (CURRENT_DATE - EMP1.EMP_BDATE) <
AVG (CURRENT_DATE - EMP.EMP_BDATE));
```

15.12.1.

**Предикат distinct**

```
SELECT DEPT.DEPT_NO
FROM DEPT, EMP EMP1, EMP EMP2
WHERE DEPT.DEPT_NO = EMP1.DEPT_NO AND
DEPT.DEPT_MNG = EMP2.EMP_NO
GROUP BY DEPT.DEPT_NO, EMP2.EMP_BDATE
HAVING (EMP2.EMP_BDATE, AVG (EMP1.EMP_SAL)) DISTINCT FROM
(SELECT EMP2.EMP_BDATE, AVG (EMP1.EMP_SAL)
FROM DEPT DEPT1, EMP EMP1, EMP EMP2
WHERE DEPT1.DEPT_NO = EMP1.DEPT_NO AND
DEPT1.DEPT_MNG = EMP2.EMP_NO AND
DEPT1.DEPT_NO <> DEPT.DEPT_NO
GROUP BY DEPT.DEPT_NO, EMP2.EMP_BDATE);
```

15.13. Найти номера отделов, которые можно отличить от любого другого отдела по дате рождения руководителя и среднему размеру зарплаты.

**Ссылки на порождаемые таблицы в разделе FROM**

В этом разделе мы приведем несколько примеров запросов, в разделе **FROM** которых содержатся выражения запросов (*ссылки на порождаемые таблицы*, см. раздел "Общие синтаксические правила построения *скалярных выражений*" лекции 13).

**Еще один способ формулировки запросов**

Прежде всего, на простом примере покажем, как использование *ссылок на порождаемые таблицы* расширяет возможности формулировки запросов.

```
SELECT MNG.DEPT_NO, MNG.MNG_NAME
FROM (SELECT DEPT.DEPT_NO, EMP.DEPT_NO, EMP_NAME, EMP_SAL
FROM DEPT, EMP
WHERE DEPT.DEPT_MNG = EMP.EMP_NO)
AS MNG (DEPT_NO_1, DEPT_NO_2, MNG_NAME, MNG_SAL)
WHERE DEPT_NO_1 = DEPT_NO_2
AND MNG_SAL = (SELECT MAX (EMP_SAL)
FROM EMP
WHERE EMP.DEPT_NO = DEPT_NO_1);
```

15.14. Найти номера отделов и имена руководителей отделов, которые числятся в тех же отделах, которыми руководят, и получают зарплату, размер которой является максимальным для служащих данного отдела.

В этом запросе порождаемая таблица **MNG** содержит по одной строке для каждого служащего, являющегося руководителем отдела. Первый столбец этой таблицы -

**DEPT\_NO\_1** - содержит номер отдела, которым руководит данный служащий. В столбце **DEPT\_NO\_1** хранятся номера отделов, в которых числятся руководители отделов, а в столбцах **EMP\_NAME** и **EMP\_SAL** содержатся имя служащего-руководителя отдела и размер его *заработной платы* соответственно.

Конечно, этот запрос можно сформулировать и без использования *ссылки на порождаемую таблицу в разделе FROM*, например, следующим образом ( [пример 15.14.1](#)):

```
SELECT DEPT.DEPT_NO, EMP.EMP_NAME
FROM DEPT, EMP
 WHERE DEPT.DEPT_MNG = EMP.EMP_NO
 AND DEPT.DEPT_NO = EMP.DEPT_NO
 AND EMP.EMP_SAL = (SELECT MAX(EMP_SAL)
 FROM EMP
 WHERE EMP.DEPT_NO = DEPT.DEPT_NO);
```

15.14.1.

А вот как можно сформулировать тот же запрос с использованием раздела **WITH** ( [пример 15.14.2](#)):

```
WITH MNG (DEPT_NO_1, DEPT_NO_2, MNG_NAME, MNG_SAL) AS
 (SELECT DEPT.DEPT_NO, EMP.DEPT_NO, EMP_NAME, EMP_SAL
 FROM DEPT, EMP
 WHERE DEPT.MNG_NO = EMP.EMP_NO),
 MAX_DEPT_SAL (MAX_SAL, DEPT_NO) AS
 (SELECT MAX (EMP_SAL), DEPT_NO
 FROM EMP
 WHERE DEPT_NO IS NOT NULL
 GROUP BY DEPT_NO)
SELECT DEPT_NO_1, MNG_NAME
FROM MNG
WHERE DEPT_NO_1 = DEPT_NO_2
 AND MNG_SAL = (SELECT MAX_SAL
 FROM MAX_DEPT_SAL
 WHERE MAX_DEPT_SAL.DEPT_NO = DEPT_NO_1);
```

15.14.2.

**Случай, в которых без порождаемых таблиц обойтись невозможно**

На самом деле, [пример 15.14](#) демонстрирует лишь возможность альтернативных формулировок запросов с использованием *ссылок на порождаемые таблицы в разделе FROM*. Но в некоторых случаях без подобных конструкций просто невозможно обойтись. Вот простой пример.

```
SELECT SUM (TOTAL_EMP), MAX_SAL
FROM (SELECT MAX (EMP_SAL), COUNT (*)
 FROM EMP
 WHERE DEPT_NO IS NOT NULL
 GROUP BY DEPT_NO) AS DEPT_MAX_SAL (MAX_SAL, TOTAL_EMP)
```

GROUP BY MAX\_SAL;

15.15. Найти общее число служащих и максимальный размер зарплаты в отделах с одинаковым максимальным размером зарплаты.

И в этом случае выражение запросов, содержащееся в разделе **FROM**, можно перенести в раздел **WITH** ( [пример 15.15.1](#)):

```
WITH DEPT_MAX_SAL (MAX_SAL, TOTAL_EMP) AS
 (SELECT MAX (EMP_SAL), COUNT (*)
 FROM EMP
 WHERE DEPT_NO IS NOT NULL
 GROUP BY DEPT_NO)
SELECT SUM (TOTAL_EMP), MAX_SAL
FROM DEPT_MAX_SAL
GROUP BY MAX_SAL;
```

15.15.1.

Здесь мы не можем обойтись "одноуровневой" конструкцией запроса, поскольку требуется двойная группировка, причем вторая группировка должна быть получена в соответствии с результатами первой. Еще один пример.

```
SELECT COUNT (*), PRO_EDATE, AVG_SAL
FROM (SELECT PRO_EDATE, AVG (EMP_SAL)
 FROM (SELECT PRO_SDATE + PRO_DURAT, PRO_NO
 FROM PRO) AS PRO1 (PRO_EDATE, PRO_NO), EMP
 WHERE PRO1.PRO_NO = EMP.PRO_NO
 GROUP BY PRO1.PRO_NO) AS PRO_AVG_SAL (PRO_EDATE, AVG_SAL)
GROUP BY PRO_EDATE, AVG_SAL;
```

15.16. Найти число проектов, дату их завершения и средний размер зарплаты служащих, участвующих в проекте, для проектов с одной и той же датой завершения и одним и тем же средним размером зарплаты служащих, участвующих в проекте.

Заметим, что выражение запросов на третьей и четвертой строках примера необходимо только по той причине, что нам требуется группировка по дате окончания проектов, соответствующий столбец в таблице *PRO* отсутствует, а в списке группировки можно использовать только имена столбцов. Для упрощения вида формулировки это выражение разумно вынести в раздел **WITH** ( [пример 15.16.1](#)):

```
WITH PRO1 (PRO_EDATE, PRO_NO) AS
 (SELECT PRO_SDATE + PRO_DURAT, PRO_NO
 FROM PRO)
SELECT COUNT (*), PRO_EDATE, AVG_SAL
FROM (SELECT PRO_EDATE, AVG (EMP_SAL)
 FROM PRO1, EMP
 WHERE PRO1.PRO_NO = EMP.PRO_NO
 GROUP BY PRO1.PRO_NO) AS PRO_AVG_SAL (PRO_EDATE, AVG_SAL)

GROUP BY PRO_EDATE, AVG_SAL;
```

### 15.16.1.

#### Более сложные конструкции оператора выборки

В этом разделе мы обсудим возможности языка *SQL*, касающиеся явного задания выражений с соединениями и *порождаемых таблиц с горизонтальной связью* (*lateral\_derived\_table*). Начнем с соединений.

#### Соединенные таблицы

В примерах предыдущей и данной лекций присутствовало много запросов с соединениями двух или более таблиц. Условия соединения задавались *предикатами сравнения* столбцов таблиц, специфицированных в разделе **FROM**, и входили в состав *логических выражений* раздела **WHERE** (или, реже, раздела **HAVING**). Поскольку на практике требуются разные виды соединений, в стандарте SQL/92 появилась альтернативная возможность спецификации соединений - *соединенная таблица* (**joined table**). Соответствующая конструкция может использоваться в разделе **FROM** выражения запросов и фактически позволяет строить выражения соединений таблиц. Синтаксические правила построения таких выражений выглядят следующим образом:

```
joined_table ::= cross_join
 | qualified_join
 | natural_join
 | union_join
cross_join ::= table_reference CROSS JOIN table_primary
qualified_join ::= table_reference [join_type] JOIN
 table_primary join_specification
natural_join ::= table_reference NATURAL [join_type]
 JOIN table_primary
union_join ::= table_reference UNION JOIN table_primary
join_type ::= INNER | { LEFT | RIGHT | FULL } [OUTER]
join_specification ::= ON conditional_expression
 | USING (column_comma_list)
```

Напомним, что синтаксические правила для **table\_reference** и **table\_primary** были показаны в лекции 13.

Как показывает сводка синтаксических правил, в SQL поддерживается много вариантов соединений. Чтобы объяснить особенности разных видов соединений на неформальном уровне, требуется очень большой объем текста с большим числом повторений. Поэтому сначала мы приведем достаточно формальное описание порядка определения заголовка и тела результирующей таблицы для всех разновидностей соединений. Фактически это описание напрямую позаимствовано из *стандарта SQL:1999* с некоторыми незначительными упрощениями. Затем мы представим ряд иллюстрирующих примеров.

#### Формальные определения

Пусть требуется выполнить некоторую операцию соединения над таблицами **table1** и **table2**. Тогда:

- Обозначим через **CP** результат выполнения запроса<sup>5</sup>

- **SELECT \***  
**FROM table1, table2**
- Если задается операция **JOIN** (или **NATURAL JOIN**) без явного указания типа соединения (**join\_type**), то по умолчанию имеется в виду **INNER JOIN** (или **NATURAL INNER JOIN**).
- Если в спецификации соединения (**join\_specification**) указано ключевое слово **ON**, то все ссылки на столбцы, встречающиеся в *условном выражении* (**conditional\_expression**), должны указывать на столбцы таблиц **table1** и **table2** или на столбцы таблиц внешнего запроса. Если в этом *условном выражении* присутствует вызов *агрегатной функции*, то *соединенная таблица* может фигурировать только в *подзапросах*, используемых в разделах **HAVING** или **SELECT** внешнего запроса, и ссылка на столбец в вызове функции должна указывать на столбец таблицы внешнего запроса.
- Для прямых соединений (**CROSS JOIN**) и всех других видов соединения, включающих раздел **ON**, заголовок результата операции совпадает с заголовком таблицы **CP**.
- Если в спецификации вида соединения присутствуют ключевые слова **NATURAL** или **USING**, то заголовок результата операции определяется следующим образом:
  - о если в спецификации вида соединения присутствует ключевое слово **NATURAL**, то будем называть *соответствующими столбцами соединения* (**corresponding join column**) все столбцы таблиц **table1** и **table2**, которые имеют в заголовках этих таблиц одинаковые имена. Если в спецификации вида соединения присутствует ключевое слово **USING**, то будем называть *соответствующими столбцами соединения* (**corresponding join column**) все столбцы таблиц **table1** и **table2**, имена которых входят в список имен столбцов раздела **USING** (эти столбцы должны быть одноименными в заголовках обеих таблиц). В обоих случаях типы данных каждой пары соответствующих столбцов должны быть совместимыми;
  - о будем называть *списком выборки соответствующих столбцов соединения* (**select\_list of corresponding join columns - SLCC**) список элементов вида **COALESCE (table1.c, table2.c) AS c\***, где **c\*** является именем *соответствующего столбца соединения*. Элементы располагаются в том порядке, в котором они появляются в заголовке таблицы **table1**. Обозначим через **SLT1** (**SLT2**) список имен столбцов таблицы **table1** (**table2**), которые не являются *соответствующими столбцами соединения*. Имена располагаются в том же порядке, в котором они появляются в заголовке соответствующей таблицы;
  - о заголовок результата совпадает с заголовком результата запроса
- **SELECT SLCC, SLT1, SLT2**  
**FROM table1, table2;**
- Набор строк результата (множество или *мультимножество*) определяется по следующим правилам. Обозначим через **T** следующие наборы строк:
  - о если видом соединения является **UNION JOIN**, то **T** - пусто;
  - о если видом соединения является **CROSS JOIN**, то **T** включает все строки, входящие в **CP**;
  - о если в спецификацию вида соединения входит раздел **ON**, то **T** включает все строки **CP**, для которых результатом вычисления условного выражения является **true**;
  - о если в спецификацию вида соединения входят разделы **NATURAL** или **USING**, и список **SLCC** не является пустым, то **T** включает все строки **CP**, для которых значения *соответствующих столбцов соединения* совпадают<sup>7</sup>;
  - о если в спецификацию вида соединения входят разделы **NATURAL** или **USING**, и список **SLCC** является пустым, то **T** включает все строки **CP**.

- Обозначим через **P1** (**P2**) набор (множество или *мультимножество*) всех строк таблицы **table1** (**table2**), каждая из которых участвует в образовании некой строки **T**.
- Обозначим через **U1** (**U2**) набор (множество или *мультимножество*) всех строк таблицы **table1** (**table2**), ни одна из которых не участвует в образовании какой-либо строки **T**.
- Обозначим через **X1** набор (множество или *мультимножество*) всех строк, образуемых из строк набора **U1** путем добавления справа подстроки из *неопределенных значений*, содержащей столько *неопределенных значений*, сколько столбцов содержит таблица **table2**. Обозначим через **X2** набор (множество или *мультимножество*) всех строк, образуемых из строк набора **U2** путем добавления слева подстроки из *неопределенных значений*, содержащей столько *неопределенных значений*, сколько столбцов содержит таблица **table1**.
- Для соединений вида **CROSS JOIN** и **INNER JOIN** пусть **S** обозначает тот же набор строк, что и **T**.
- Для соединений вида **LEFT OUTER JOIN** пусть **S** обозначает набор строк, являющийся результатом выражения запросов
  - **SELECT \* FROM T**
  - **UNION ALL**  
**SELECT \* FROM X1;**
- Для соединений вида **RIGHT OUTER JOIN** пусть **S** обозначает набор строк, являющийся результатом выражения запросов
  - **SELECT \* FROM T**
  - **UNION ALL**  
**SELECT \* FROM X2;**
- Для соединений вида **FULL OUTER JOIN** пусть **S** обозначает набор строк, являющийся результатом выражения запросов
  - **SELECT \* FROM T**
  - **UNION ALL**
  - **SELECT \* FROM X1**
  - **UNION ALL**  
**SELECT \* FROM X2;**
- Для соединений вида **UNION JOIN** пусть **S** обозначает набор строк, являющийся результатом выражения запросов
  - **SELECT \* FROM X1**
  - **UNION ALL**  
**SELECT \* FROM X2;**
- Если в спецификации вида соединения присутствуют ключевые слова **NATURAL** или **USING**, то результат операции совпадает с результатом выражения запросов
  - **SELECT SLCC, SLT1, SLT2**  
**FROM S;**
- Во всех остальных случаях результат операции совпадает с **S**.

#### **Примеры соединений разного вида**

Основное назначение приводимых ниже примеров состоит не в том, чтобы продемонстрировать практическую значимость разнообразных соединений, а лишь в том, чтобы помочь в них разобраться.<sup>8</sup> Поэтому мы будем использовать упрощенные и формальные таблицы и показывать заголовки и тела результирующих таблиц.

Итак, пусть имеются таблицы **table1** (a1, a2, c1, c2) и **table2** (b1, b2, c1, c2) со следующими телами:

| table1 |      |      |      |
|--------|------|------|------|
| a1     | a2   | c1   | c2   |
| 1      | 1    | 1    | 1    |
| 1      | 1    | 2    | 3    |
| 1      | 1    | 2    | 3    |
| 2      | 3    | 4    | NULL |
| 3      | NULL | NULL | 5    |

| table2 |      |      |    |
|--------|------|------|----|
| b1     | b2   | c1   | c2 |
| 1      | 1    | 1    | 1  |
| 1      | 2    | 2    | 3  |
| 3      | 3    | 2    | 3  |
| 4      | 4    | 4    | 4  |
| 3      | NULL | NULL | 5  |
| 3      | NULL | NULL | 5  |

Обозначим через **JR** таблицу, являющуюся результатом соединения. Тогда для операции **table1 INNER JOIN table2 ON a1=b1 AND a2<b2** (внутреннее соединение по условию) тело **JR** будет следующим:

| JR |    |           |           |    |    |           |           |
|----|----|-----------|-----------|----|----|-----------|-----------|
| a1 | a2 | table1.c1 | table1.c2 | b1 | b2 | table2.c1 | table2.c2 |
| 1  | 1  | 1         | 1         | 1  | 2  | 2         | 3         |
| 1  | 1  | 2         | 3         | 1  | 2  | 2         | 3         |
| 1  | 1  | 2         | 3         | 1  | 2  | 2         | 3         |

Строки-дубликаты появились в **JR**, поскольку в первом операнде присутствовали строки-дубликаты, удовлетворяющие условию соединения.

Результатом операции **table1 INNER JOIN table2 USING (c2)** ( *внутреннее соединение по совпадению значений указанных одноименных столбцов* ) будет следующая таблица.

| JR |      |           |    |    |      |           |
|----|------|-----------|----|----|------|-----------|
| a1 | a2   | table1.c1 | c2 | b1 | b2   | table2.c1 |
| 1  | 1    | 1         | 1  | 1  | 1    | 1         |
| 1  | 1    | 2         | 3  | 1  | 2    | 2         |
| 1  | 1    | 2         | 3  | 3  | 3    | 2         |
| 1  | 1    | 2         | 3  | 1  | 2    | 2         |
| 1  | 1    | 2         | 3  | 3  | 3    | 2         |
| 3  | NULL | NULL      | 5  | 3  | NULL | NULL      |
| 3  | NULL | NULL      | 5  | 3  | NULL | NULL      |

Результат операции **table1 INNER JOIN table2 USING (c1,c2)**:

| JR |    |    |    |    |    |
|----|----|----|----|----|----|
| a1 | a2 | c1 | c2 | b1 | b2 |
| 1  | 1  | 1  | 1  | 1  | 1  |
| 1  | 1  | 2  | 3  | 1  | 2  |
| 1  | 1  | 2  | 3  | 3  | 3  |
| 1  | 1  | 2  | 3  | 1  | 2  |
| 1  | 1  | 2  | 3  | 3  | 3  |

Такой же результат будет получен при выполнении операции `table1 NATURAL INNER JOIN table2` (*естественное внутреннее соединение*). Более того, для произвольных таблиц `table1` и `table2` результаты операций `table1 INNER JOIN table2 USING (c1, c2, ...cn)` и `table1 INNER NATURAL JOIN table2` совпадают в том и только в том случае, когда список имен столбцов `c1, c2, ... cn` включает все имена столбцов, общие для таблиц `table1` и `table2`.

Результатом операции `table1 LEFT OUTER JOIN table2 ON a1=b1 AND a2<b2` (*левое внешнее соединение по условию >*) будет следующая таблица:

| JR |    |           |           |      |      |           |           |
|----|----|-----------|-----------|------|------|-----------|-----------|
| a1 | a2 | table1.c1 | table1.c2 | b1   | b2   | table2.c1 | table2.c2 |
| 1  | 1  | 1         | 1         | 1    | 2    | 2         | 3         |
| 1  | 1  | 2         | 3         | 1    | 2    | 2         | 3         |
| 1  | 1  | 2         | 3         | 1    | 2    | 2         | 3         |
| 2  | 3  | 4         | NULL      | NULL | NULL | NULL      | NULL      |

|   |      |      |   |      |      |      |      |
|---|------|------|---|------|------|------|------|
| 3 | NULL | NULL | 5 | NULL | NULL | NULL | NULL |
|---|------|------|---|------|------|------|------|

Как видно, в результате *левого внешнего соединения* сохраняются все данные первого (левого) операнда.

Результатом операции `table1 RIGHT OUTER JOIN table2 ON a1=b1 AND a2<b2` (*правое внешнее соединение по условию*) будет следующая таблица:

| JR   |      |           |           |    |      |           |           |
|------|------|-----------|-----------|----|------|-----------|-----------|
| a1   | a2   | table1.c1 | table1.c2 | b1 | b2   | table2.c1 | table2.c2 |
| 1    | 1    | 1         | 1         | 1  | 2    | 2         | 3         |
| 1    | 1    | 2         | 3         | 1  | 2    | 2         | 3         |
| 1    | 1    | 2         | 3         | 1  | 2    | 2         | 3         |
| NULL | NULL | NULL      | NULL      | 1  | 1    | 1         | 1         |
| NULL | NULL | NULL      | NULL      | 3  | 3    | 2         | 3         |
| NULL | NULL | NULL      | NULL      | 4  | 4    | 4         | 4         |
| NULL | NULL | NULL      | NULL      | 3  | NULL | NULL      | 5         |
| NULL | NULL | NULL      | NULL      | 3  | NULL | NULL      | 5         |

Как видно, в результате *правого внешнего соединения* сохраняются все данные второго (правого) операнда.

Результатом операции `table1 FULL OUTER JOIN table2 ON a1=b1 AND a2<b2` (*полное внешнее соединение по условию*) будет следующая таблица:

| JR |  |  |  |  |  |  |  |
|----|--|--|--|--|--|--|--|
|----|--|--|--|--|--|--|--|

| a1   | a2   | table1.c1 | table1.c2 | b1   | b2   | table2.c1 | table2.c2 |
|------|------|-----------|-----------|------|------|-----------|-----------|
| 1    | 1    | 1         | 1         | 1    | 2    | 2         | 3         |
| 1    | 1    | 2         | 3         | 1    | 2    | 2         | 3         |
| 1    | 1    | 2         | 3         | 1    | 2    | 2         | 3         |
| 2    | 3    | 4         | NULL      | NULL | NULL | NULL      | NULL      |
| 3    | NULL | NULL      | 5         | NULL | NULL | NULL      | NULL      |
| NULL | NULL | NULL      | NULL      | 1    | 1    | 1         | 1         |
| NULL | NULL | NULL      | NULL      | 3    | 3    | 2         | 3         |
| NULL | NULL | NULL      | NULL      | 4    | 4    | 4         | 4         |
| NULL | NULL | NULL      | NULL      | 3    | NULL | NULL      | 5         |
| NULL | NULL | NULL      | NULL      | 3    | NULL | NULL      | 5         |

Как видно, в результате *полного внешнего соединения* сохраняются данные обоих операндов. Кстати, *полное внешнее соединение* иногда называют еще *симметричным внешним соединением*. Очевидно, что все операции внутреннего соединения и операция *полного внешнего соединения* коммутативны, а операции левого и правого соединения коммутативными не являются.

Результатом операции `table1 LEFT OUTER JOIN table2 USING (c2)` (*левое внешнее соединение по совпадению значений указанных одноименных столбцов* >) будет следующая таблица:

| JR |    |           |    |    |    |           |
|----|----|-----------|----|----|----|-----------|
| a1 | a2 | table1.c1 | c2 | b1 | b2 | table2.c1 |

|   |      |      |      |      |      |      |
|---|------|------|------|------|------|------|
| 1 | 1    | 1    | 1    | 1    | 1    | 1    |
| 1 | 1    | 2    | 3    | 1    | 2    | 2    |
| 1 | 1    | 2    | 3    | 3    | 3    | 2    |
| 1 | 1    | 2    | 3    | 1    | 2    | 2    |
| 1 | 1    | 2    | 3    | 3    | 3    | 2    |
| 3 | NULL | NULL | 5    | 3    | NULL | NULL |
| 3 | NULL | NULL | 5    | 3    | NULL | NULL |
| 2 | 3    | 4    | NULL | NULL | NULL | NULL |

Результатом операции `table1 RIGHT OUTER JOIN table2 USING (c2)` (*правое внешнее соединение по совпадению значений указанных одноименных столбцов*) будет следующая таблица:

| JR |    |           |    |    |    |           |
|----|----|-----------|----|----|----|-----------|
| a1 | a2 | table1.c1 | c2 | b1 | b2 | table2.c1 |
| 1  | 1  | 1         | 1  | 1  | 1  | 1         |
| 1  | 1  | 2         | 3  | 1  | 2  | 2         |
| 1  | 1  | 2         | 3  | 3  | 3  | 2         |
| 1  | 1  | 2         | 3  | 1  | 2  | 2         |
| 1  | 1  | 2         | 3  | 3  | 3  | 2         |

|      |      |      |   |   |      |      |
|------|------|------|---|---|------|------|
| 3    | NULL | NULL | 5 | 3 | NULL | NULL |
| 3    | NULL | NULL | 5 | 3 | NULL | NULL |
| NULL | NULL | NULL | 4 | 4 | 4    | 4    |

Результатом операции `table1 FULL OUTER JOIN table2 USING (c2)` (полное внешнее соединение по совпадению значений указанных одноименных столбцов) будет следующая таблица:

| JR   |      |           |      |      |      |           |
|------|------|-----------|------|------|------|-----------|
| a1   | a2   | table1.c1 | c2   | b1   | b2   | table2.c1 |
| 1    | 1    | 1         | 1    | 1    | 1    | 1         |
| 1    | 1    | 2         | 3    | 1    | 2    | 2         |
| 1    | 1    | 2         | 3    | 3    | 3    | 2         |
| 1    | 1    | 2         | 3    | 1    | 2    | 2         |
| 1    | 1    | 2         | 3    | 3    | 3    | 2         |
| 3    | NULL | NULL      | 5    | 3    | NULL | NULL      |
| 3    | NULL | NULL      | 5    | 3    | NULL | NULL      |
| 2    | 3    | 4         | NULL | NULL | NULL | NULL      |
| NULL | NULL | NULL      | 4    | 4    | 4    | 4         |

Результатом операции `table1 LEFT OUTER JOIN table2 USING (c2, c1)` (и операции `table1 NATURAL LEFT OUTER JOIN table2` - естественное левое внешнее соединение) будет следующая таблица:

| JR |      |      |      |      |      |
|----|------|------|------|------|------|
| a1 | a2   | c1   | c2   | b1   | b2   |
| 1  | 1    | 1    | 1    | 1    | 1    |
| 1  | 1    | 2    | 3    | 1    | 2    |
| 1  | 1    | 2    | 3    | 3    | 3    |
| 1  | 1    | 2    | 3    | 1    | 2    |
| 1  | 1    | 2    | 3    | 3    | 3    |
| 2  | 3    | 4    | NULL | NULL | NULL |
| 3  | NULL | NULL | 5    | NULL | NULL |

Результатом операции `table1 RIGHT OUTER JOIN table2 USING (c2, c1)` (и операции `table1 NATURAL RIGHT OUTER JOIN table2` - *естественное правое внешнее соединение* ) будет следующая таблица:

| JR |    |    |    |    |    |
|----|----|----|----|----|----|
| a1 | a2 | c1 | c2 | b1 | b2 |
| 1  | 1  | 1  | 1  | 1  | 1  |
| 1  | 1  | 2  | 3  | 1  | 2  |
| 1  | 1  | 2  | 3  | 3  | 3  |
| 1  | 1  | 2  | 3  | 1  | 2  |

|      |      |      |   |   |      |
|------|------|------|---|---|------|
| 1    | 1    | 2    | 3 | 3 | 3    |
| NULL | NULL | 4    | 4 | 4 | 4    |
| NULL | NULL | NULL | 5 | 3 | NULL |
| NULL | NULL | NULL | 5 | 3 | NULL |

Результатом операции `table1 FULL OUTER JOIN table2 USING (c2, c1)` (и операции `table1 NATURAL FULL OUTER JOIN table2` - *естественное полное внешнее соединение* ) будет следующая таблица:

| JR   |      |      |      |      |      |
|------|------|------|------|------|------|
| a1   | a2   | c1   | c2   | b1   | b2   |
| 1    | 1    | 1    | 1    | 1    | 1    |
| 1    | 1    | 2    | 3    | 1    | 2    |
| 1    | 1    | 2    | 3    | 3    | 3    |
| 1    | 1    | 2    | 3    | 1    | 2    |
| 1    | 1    | 2    | 3    | 3    | 3    |
| 2    | 3    | 4    | NULL | NULL | NULL |
| 3    | NULL | NULL | 5    | NULL | NULL |
| NULL | NULL | 4    | 4    | 4    | 4    |
| NULL | NULL | NULL | 5    | 3    | NULL |

|      |      |      |   |   |      |
|------|------|------|---|---|------|
| NULL | NULL | NULL | 5 | 3 | NULL |
|------|------|------|---|---|------|

Наконец, результатом операции **table1 UNION JOIN table2** (соединение объединением) будет следующая таблица:

| JR   |      |           |           |      |      |           |           |
|------|------|-----------|-----------|------|------|-----------|-----------|
| a1   | a2   | table1.c1 | table1.c2 | b1   | b2   | table2.c1 | table2.c2 |
| 1    | 1    | 1         | 1         | NULL | NULL | NULL      | NULL      |
| 1    | 1    | 2         | 3         | NULL | NULL | NULL      | NULL      |
| 1    | 1    | 2         | 3         | NULL | NULL | NULL      | NULL      |
| 2    | 3    | 4         | NULL      | NULL | NULL | NULL      | NULL      |
| 3    | NULL | NULL      | 5         | NULL | NULL | NULL      | NULL      |
| NULL | NULL | NULL      | NULL      | 1    | 1    | 2         | 3         |
| NULL | NULL | NULL      | NULL      | 1    | 2    | 2         | 3         |
| NULL | NULL | NULL      | NULL      | 3    | 3    | 2         | 3         |
| NULL | NULL | NULL      | NULL      | 4    | 4    | 4         | 4         |
| NULL | NULL | NULL      | NULL      | 3    | NULL | NULL      | 5         |
| NULL | NULL | NULL      | NULL      | 3    | NULL | NULL      | 5         |

### Примеры запросов с использованием соединенных таблиц

Мы приведем всего пару примеров, чтобы проиллюстрировать формулировки запросов, в разделе **FROM** которых используются ссылки на *соединенные таблицы*, т. е. выражения соединений.

```

SELECT DEPT.DEPT_NO, EMP1.EMP_NAME, COUNT(*), MIN(EMP2.EMP_SAL),
 MAX(EMP2.EMP_SAL), AVG(EMP2.EMP_SAL)
FROM (DEPT NATURAL INNER JOIN EMP AS EMP2)
 INNER JOIN EMP AS EMP1 ON DEPT.DEPT_MNG = EMP1.EMP_NO
GROUP BY DEPT.DEPT_NO, EMP1.EMP_NAME;

```

15.17. Для каждого отдела найти его номер, имя руководителя, число служащих, минимальный, максимальный и средний размеры зарплаты служащих (еще одна формулировка запроса из примера 15.4).

```

SELECT EMP1.EMP_NO, EMP2.EMP_NAME
FROM (EMP AS EMP1 NATURAL INNER JOIN DEPT)
 INNER JOIN EMP AS EMP2 ON DEPT.DEPT_MNG = EMP2.EMP_NO
WHERE EMP1.EMP_SAL > 30000.00;

```

15.18. Найти номера служащих и имена их начальников отделов для служащих, размер зарплаты которых больше 30000 руб.

Можно обойтись вообще без раздела **WHERE**, если пожертвовать "естественностью" первого соединения ( [пример 15.17.1](#)):

```

SELECT EMP1.EMP_NO, EMP2.EMP_NAME
FROM (EMP AS EMP1 INNER JOIN DEPT
 ON EMP1.DEPT_NO = DEPT.DEPT_NO AND
 EMP1.EMP_SAL > 30000.00)
 INNER JOIN EMP AS EMP2 ON DEPT.MNG = EMP2.EMP_NO;

```

15.17.1.

Возможности *соединенных таблиц* открывают широкий простор для воображения, но не будем увлекаться и ограничимся приведенными простыми примерами.

### Порождаемые таблицы с горизонтальной связью (**lateral\_derived\_table**)

Во всех вариантах построения запросов, обсуждавшихся ранее в этой и предыдущей лекциях, оставалась действующей общая семантика выполнения запроса: на первом шаге вычисляется расширенное *декартово произведение* таблиц, специфицированных в списке раздела **FROM**. Это остается верным и для случаев порождаемых и *соединенных таблиц* - *вычисление выражения* запросов или выражения соединений соответственно производится как подшаг вычисления раздела **FROM**. Однако в SQL имеется один специальный случай спецификации ссылки на таблицу ( **table\_reference** ), который, вообще говоря, изменяет семантику раздела **FROM**. В этом подразделе мы кратко рассмотрим этот специальный случай.

Как показывают синтаксические правила, приведенные в лекции 13, один из возможных способов спецификации ссылки на таблицу состоит в следующем:

```

table_reference ::= LATERAL (query_expression)
 [[AS] correlation_name
 [(derived_column_list)]]

```

Таблица, ссылка на которую специфицируется таким образом, называется *порождаемой таблицей с горизонтальной связью*<sup>9</sup> ( **lateral\_derived\_table** ; для краткости будем называть

такие таблицы LD-таблицами). Отличие LD-таблицы от обычной порождаемой таблицы состоит в том, что в выражении запросов LD-таблицы разрешается использовать ссылки на столбцы таблиц, специфицированных ранее в разделе **FROM** (т. е. таких таблиц, ссылки на которые содержатся в списке раздела **FROM** слева от ссылки на данную LD-таблицу).<sup>10</sup> Покажем на примере, каким образом наличие в списке раздела **FROM** ссылки на LD-таблицу меняет семантику этого раздела.

Предположим, что раздел **FROM** имеет вид **FROM T1, T2**, причем таблица **T2** является LD-таблицей. Обозначим соответствующее выражение запросов через **Q2**. Тогда таблица **T**, являющаяся результатом раздела **FROM**, будет вычисляться следующим образом. Последовательно, строка за строкой просматривается таблица **T1**. Пусть **s1** является очередной строкой **T1**. Тогда в **Q2** все ссылки на столбцы вида **T1.ck**, где **ck** - имя некоторого столбца **T1**, заменяются значением **s1.ck**, и вычисляется полученное таким образом выражение запросов. Обозначим результирующую таблицу этого выражения через **T2<sub>s1</sub>**. Обозначим через **T12<sub>s1</sub>** таблицу, являющуюся результатом расширенного декартова произведения **s1 CROSS JOIN T2<sub>s1</sub>**. Таблица **T** получается путем объединения с сохранением дубликатов таблиц **T12<sub>s1</sub>**, полученных для всех строк **s1** таблицы **T1**.

Видимо, наиболее важным (хотя и не единственным) частным случаем применения LD-таблицы является тот случай, когда в результате выполнения раздела **FROM** формируется соединение таблиц. Многие из формулировок запросов, приводившихся в этой лекции в качестве примеров, можно переформулировать с использованием данного механизма. Приведем лишь один простой пример.

```
SELECT EMP.EMP_NO
FROM DEPT, LATERAL
 (SELECT EMP1_SAL
 FROM EMP EMP1
 WHERE EMP1.EMP_NO = DEPT.DEPT_MNG),
LATERAL
 (SELECT EMP_NO
 FROM EMP
 WHERE EMP_SAL = EMP1_SAL AND
 EMP.EMP_NO <> DEPT.DEPT_MNG);
```

15.19. Найти номера служащих, не являющихся руководителями отделов и получающих зарплату, размер которой равен размеру зарплаты какого-либо руководителя отдела (еще одна формулировка запроса из примера 14.10 из лекции 14).

## Заключение

Думаю, что теперь читатели в состоянии в полной мере оценить *мощность*, разнообразие и *избыточность* средств языка *SQL*, предназначенных для формулировки запросов на выборку данных. Конечно, язык *SQL* (покрайней мере, ту часть *SQL*, которая обсуждается в этом курсе) нельзя считать языком программирования, но написание сложных запросов сродни программированию. И нельзя сказать, что *SQL* каким-либо образом дисциплинирует это "*программирование*". По всей видимости, в общем случае никто не может сказать, какая из формулировок одного и того же запроса является более правильной, это дело вкуса.

Зачастую десять студентов, одновременно формулирующих на *SQL* один и тот же *запрос* к одной и той же базе данных, выдают десять разных правильных решений. Один человек предпочитает формулировки запросов в классическом стиле, другой использует выражения запросов в разделе **FROM**, третий пытается сосредоточить все условия выборки в разделе **HAVING**. Люди с алгебраическими наклонностями предпочитают использовать выражения соединений. Приходилось встречать и формулировки со сложными вложенными подзапросами в списке выборки раздела **SELECT**.

Конечно, теоретически *компилятор SQL* должен быть в состоянии распознать все эквивалентные формулировки одного и того же запроса и выработать для всех них один и тот же наиболее эффективный *план выполнения*. Но чем больше разнообразие возможных формулировок, тем сложнее эта задача. Отсюда практический совет: не злоупотребляйте сложностью формулировки запроса. Полагайтесь на интуицию (и имеющиеся представления об особенностях используемой системы) и формулируйте *запрос* как можно проще.

И еще один практический совет. При формулировке запроса никогда не пользуйтесь имеющимися у вас данными о текущем состоянии *базы данных*, полагайтесь только на *метаданные* схемы *базы данных*. В противном случае вы сможете сформулировать *запрос*, выдающий в данный момент правильный результат, но этот *запрос* не будет эквивалентен никакому запросу, выдающему *правильный ответ* при любом состоянии *базы данных*.

## Лабораторная работа 6. Средства формулировки аналитических и рекурсивных запросов

### Введение

Две темы, которым посвящается эта лекция, касаются сравнительно новых возможностей оператора **SELECT** языка *SQL*, впервые появившихся в *стандарте SQL:1999* и открывающих возможность использования языка в приложениях, для которых ранее он не был приспособлен. Речь идет о возможностях *аналитических* и *рекурсивных запросов*. Эти темы логически не связаны, их объединяет лишь то, что соответствующие средства очень громоздки и не всегда легко понимаются. В данной краткой лекции мы не стремимся привести полное описание возможностей, специфицированных в стандарте *SQL*. Наша цель состоит лишь в том, чтобы в общих чертах описать подход *SQL* в указанных направлениях.

В аналитических приложениях обычно требуются не детальные данные, непосредственно хранящиеся в базе данных, а некоторые их обобщения, агрегаты. Например, аналитика интересуется не *заработная плата* конкретного человека в конкретное время, а изменение *заработной платы* некоторой категории людей в течение определенного промежутка времени. Если пользоваться терминологией *SQL*, то типичный *запрос* к базе данных со стороны аналитического приложения содержит раздел **GROUP BY** и вызовы *агрегатных функций*. Хотя в этом курсе мы почти не касаемся вопросов реализации *SQL*-ориентированных *СУБД*, из общих соображений должно быть понятно, что запросы с разделом **GROUP BY** в общем случае являются "трудными" для *СУБД*, поскольку для группирования таблицы, вообще говоря, требуется *внешняя сортировка*.

В системах баз данных, специально спроектированных в расчете на аналитические приложения, проблему обычно решают за счет явного избыточного хранения *агрегированных данных* (т.е. результатов вызова *агрегатных функций*).

Конечно, для этого требуется *динамическая* корректировка хранимых агрегатных значений при изменении детальных данных, но для таких специализированных баз данных это не слишком обременительно, поскольку аналитические *базы данных* обновляются сравнительно редко.

Однако далеко не каждое предприятие может позволить себе одновременно поддерживать оперативную базу данных для работы обычных приложений оперативной обработки транзакций (*OLTP*), таких, как бухгалтерские, кадровые и другие приложения, и аналитическую базу данных для приложений *оперативной аналитической обработки* (*OLAP*). Приходится выполнять аналитические приложения над детальными оперативными базами данных, и эти приложения обращаются к *СУБД* с многочисленными трудоемкими запросами с разделами **GROUP BY** и вызовами *агрегатных функций*.

Разработчики *стандарта языка SQL* старались одновременно решить две задачи: сократить число запросов, требуемых в аналитических приложениях, и добиться снижения стоимости запросов с разделом **GROUP BY**, обеспечивающих требуемые суммарные данные. В этой лекции мы обсудим наиболее важные, с нашей точки зрения, конструкции языка *SQL*, облегчающие формулировку, выполнение и использование результатов *аналитических запросов*: разделы **GROUP BY ROLLUP** и **GROUP BY CUBE** и новую *агрегатную функцию* **GROUPING**, позволяющую правильно трактовать результаты *аналитических запросов* при наличии *неопределенных значений*.

Традиционно язык *SQL* никогда не обладал возможностью формулировки *рекурсивных запросов*, где под *рекурсивным запросом* (упрощенно говоря) мы понимаем запрос к таблице, которая сама каким-либо образом изменяется при выполнении этого запроса. Напомню, что это заложено в базовую семантику *оператора SQL*: до выполнения раздела **WHERE** результат раздела **FROM** должен быть полностью вычислен.

Однако разработчикам приложений часто приходится решать задачи, для которых недостаточно традиционных средств формулировки запросов языка *SQL*: например, нахождение маршрута движения между двумя заданными географическими точками, определения общего набора комплектующих для сбора некоторого агрегата и т.д. Компании-производители *SQL-ориентированных СУБД* пытались удовлетворять такие потребности за счет *частных решений*, обладающих ограниченными рекурсивными свойствами, но до появления стандарта *SQL:1999* общие стандартизованные средства отсутствовали.

Следует отметить и некоторое давление на *SQL-сообщество* со стороны сообщества логических систем баз данных. На основе языка *логического программирования Prolog* был разработан язык *реляционных баз данных Datalog*, обеспечивающий все необходимые средства для обычной работы с базами данных наряду с развитыми возможностями *рекурсивных запросов*. Требовался адекватный ответ со стороны разработчиков стандарта *SQL*.

Компромиссное (не слишком красивое) решение для введения рекурсии в *SQL* было найдено на основе введения *раздела WITH* в *выражение* запроса. Только в этом разделе допускается как *линейная*, так и *взаимная рекурсия* между вводимыми порождаемыми таблицами. При этом только для *линейной рекурсии* обеспечиваются дополнительные возможности управления порядком вычисления *рекурсивно определенной* порождаемой таблицы и контроля отсутствия циклов. Следует заметить, что при чтении стандарта временами возникает впечатление, что его авторы сами не до конца еще осознали всех

возможных последствий, к которым может привести использование введенных конструкций. Я думаю, что в следующих версиях стандарта следует ожидать уточнений и/или ограничений использования названных конструкций. В связи с этим в данной лекции мы ограничиваемся общими определениями рекурсивных конструкций языка *SQL* и обсуждением простого случая *рекурсивного запроса*.

## Возможности формулирования аналитических запросов

*Аналитическими запросами к базе данных* принято называть запросы, сводные (агрегатные) результаты которых вычисляются над детальными данными, хранящимися в таблицах базы данных. В этом смысле любой запрос на языке *SQL*, результат которого основан на вычислении *агрегатных функций*, можно назвать аналитическим. Характерная особенность *аналитических запросов* состоит в том, что, как правило, они применяются к большим по объему базам данных, и выполнение таких *запросов* вызывает существенные накладные *расходы СУБД*.

В этом курсе мы не будем подробно обсуждать возможности языка *SQL*, предназначенные для поддержки *оперативной аналитической обработки баз данных (OLAP - on-line analytical processing)*. Рассмотрим только самые *основные средства*, опираясь на простые примеры. Для этих примеров предположим, что *таблица EMP* содержит следующий набор строк (покажем содержимое только тех столбцов, которые потребуются в примерах, причем для простоты будем считать, что в столбце *EMP\_DATE* содержится не полная дата, а только год рождения служащего):

| <i>EMP</i>    |                |                  |                |
|---------------|----------------|------------------|----------------|
| <i>EMP_NO</i> | <i>DEPT_NO</i> | <i>EMP_BDATE</i> | <i>EMP_SAL</i> |
| 2440          | 1              | 1950             | 15000.00       |
| 2441          | 1              | 1950             | 16000.00       |
| 2442          | 1              | 1960             | 14000.00       |
| 2443          | 1              | 1960             | 19000.00       |
| 2444          | 2              | 1950             | 17000.00       |
| 2445          | 2              | 1950             | 16000.00       |
| 2446          | 2              | 1960             | 14000.00       |

|      |   |      |          |
|------|---|------|----------|
| 2447 | 2 | 1960 | 20000.00 |
| 2448 | 3 | 1950 | 18000.00 |
| 2449 | 3 | 1950 | 13000.00 |
| 2450 | 3 | 1960 | 21000.00 |
| 2451 | 3 | 1960 | 22000.00 |

Представим себе, что для проведения анализа требуется узнать максимальный размер зарплаты на всем предприятии, максимальный размер зарплаты в каждом отделе и максимальный размер зарплаты служащих каждой возрастной категории каждого отдела. Если пользоваться стандартными средствами языка *SQL*, обсуждавшимися ранее в предложенном курсе, то для получения этих данных потребуется три запроса:

```
SELECT MAX (EMP_SAL) AS MAX_ENT_SAL
 FROM EMP;
SELECT DEPT_NO, MAX (EMP_SAL) AS MAX_DEP_SAL
 FROM EMP
 GROUP BY DEPT_NO;
SELECT DEPT_NO, EMP_BDATE, MAX (EMP_SAL)
 AS MAX_DEP_BDATE_SAL
 FROM EMP
 GROUP BY DEPT_NO, EMP_BDATE;
```

При выполнении запросов будут получены следующие результирующие таблицы:

| MAX_ENT_SAL |             |
|-------------|-------------|
| 22000.00    |             |
| DEPT_NO     | MAX_DEP_SAL |
| 1           | 19000.00    |
| 2           | 20000.00    |

| 3       |           | 22000.00          |
|---------|-----------|-------------------|
| DEPT_NO | EMP_BDATE | MAX_DEP_BDATE_SAL |
| 1       | 1950      | 16000.00          |
| 1       | 1960      | 19000.00          |
| 2       | 1950      | 17000.00          |
| 2       | 1960      | 20000.00          |
| 3       | 1950      | 18000.00          |
| 3       | 1960      | 22000.00          |

#### Раздел GROUP BY ROLLUP

Эти же результаты можно получить при выполнении единственного запроса, если в его формулировке использовать специальный вид группировки **ROLLUP** ([пример 16.1](#)):

```
SELECT DEPT_NO, EMP_BDATE, MAX (EMP_SAL) AS MAX_SAL
FROM EMP
GROUP BY ROLLUP (DEPT_NO, EMP_BDATE);
```

Пример 16.1.

Сначала покажем, как будет выглядеть результирующая таблица этого запроса, а потом приведем развернутое пояснение действия новой конструкции. В результате выполнения запроса будет получена таблица, показанная на [рис.16.1](#).

Как видно, в столбце **MAX\_SAL** первой строки<sup>1</sup> результирующей таблицы находится максимальное значение зарплаты служащих на всем предприятии. Столбцы **DEPT\_NO** и **EMP\_BDATE** в этой строке содержат *неопределенное значение*, поскольку значение **MAX\_SAL** не привязано к каким-либо отделу и возрастной категории. В столбце **MAX\_SAL** следующих трех строк находятся максимальные значения зарплаты служащих отделов с номерами 1, 2 и 3 соответственно, что показывают значения столбца **DEPT\_NO**. Столбец **EMP\_BDATE** в этих строках содержит *неопределенное значение*, поскольку значение **MAX\_SAL** не привязано к какой-либо возрастной категории. Наконец, в столбце **MAX\_SAL** в последних шести строках содержатся максимальные значения зарплаты служащих каждой возрастной категории каждого отдела, что показывают значения столбцов **DEPT\_NO** и **EMP\_BDATE**, которые теперь содержат соответствующий номер отдела и год рождения служащих.

| DEPT_NO | EMP_BDATE | MAX_SAL  |
|---------|-----------|----------|
| NULL    | NULL      | 22000.00 |
| 1       | NULL      | 19000.00 |
| 2       | NULL      | 20000.00 |
| 3       | NULL      | 22000.00 |
| 1       | 1950      | 16000.00 |
| 1       | 1960      | 19000.00 |
| 2       | 1950      | 17000.00 |
| 2       | 1960      | 20000.00 |
| 3       | 1950      | 18000.00 |
| 3       | 1960      | 22000.00 |

**Рис. 16.1.** Результат запроса с разделом GROUP BY ROLLUP

В общем случае пусть раздел группировки запроса имеет вид **GROUP BY ROLLUP ( cname<sub>1</sub>, cname<sub>2</sub>, ... , cname<sub>n</sub> )**, где cname<sub>i</sub> ( i = 1, 2, ... , n ) - имя столбца таблицы-результата раздела **FROM** запроса. Пусть в списке выборки используются вызовы *агрегатных функций* AGG<sub>1</sub>, AGG<sub>2</sub>, ... , AGG<sub>m</sub> над значениями столбцов, не входящих в список группировки, а также имена столбцов cname<sub>1</sub>, cname<sub>2</sub>, ... , cname<sub>n</sub>. Тогда запрос выполняется следующим образом. Первая строка результата (первый набор строк результирующей таблицы) производится таким образом, как если бы в запросе вообще отсутствовал раздел **GROUP BY**, т.е. агрегатные функции AGG<sub>1</sub>, AGG<sub>2</sub>, ... , AGG<sub>m</sub> вычисляются над значениями всех строк таблицы. Значением столбцов cname<sub>1</sub>, cname<sub>2</sub>, ... , cname<sub>n</sub> в этой строке является NULL. (i+1)-й набор строк результата формируется так, как если бы раздел группировки запроса имел вид **GROUP BY ( cname<sub>1</sub>, cname<sub>2</sub>, ... , cname<sub>i</sub> ) ( 1≤i<n )**. Во всех этих строках значением столбцов cname<sub>(i+1)</sub>, ... , cname<sub>n</sub> является NULL. Наконец, (n+1)-й набор строк результата формируется так, как если бы раздел группировки запроса имел вид **GROUP BY ( cname<sub>1</sub>, cname<sub>2</sub>, ... , cname<sub>n</sub> )**.

Может показаться, что запросы, содержащие раздел **GROUP BY ROLLUP**, настолько сложны, что их выполнение будет занимать чрезмерно большое время. Это ощущение является ложным. В действительности, при выполнении запросов с обычной группировкой вида **GROUP BY cname<sub>1</sub>, cname<sub>2</sub>, ... , cname<sub>n</sub>**, как правило, последовательно выполняется сортировка строк таблицы-результата раздела **FROM** в соответствии со значениями столбца cname<sub>1</sub>, затем - в соответствии со значениями столбца cname<sub>2</sub> и т. д., и в заключение - сортировка в соответствии со значениями столбца cname<sub>n</sub>. Во время выполнения каждой сортировки можно заодно вычислять значения *агрегатных функций*. Так что стоимость выполнения запроса, содержащего раздел **GROUP BY ROLLUP**, лишь незначительно отличается от стоимости выполнения запроса с обычной группировкой.

### Агрегатная функция GROUPING

Обсудим теперь один более тонкий вопрос. Как говорилось в лекции 12, *определение столбцов DEPT\_NO и EMP\_BDATE* таблицы **EMP** допускает появление в этих столбцах *неопределенных значений*. Поэтому тело таблицы **EMP** могло бы иметь, например, следующий вид:

| <i>EMP</i> |         |           |          |
|------------|---------|-----------|----------|
| EMP_NO     | DEPT_NO | EMP_BDATE | EMP_SAL  |
| 2440       | 1       | 1950      | 15000.00 |
| 2441       | 1       | 1950      | 16000.00 |
| 2442       | 1       | 1960      | 14000.00 |
| 2443       | 1       | 1960      | 19000.00 |
| 2452       | 1       | NULL      | 15000.00 |
| 2453       | 1       | NULL      | 17000.00 |
| 2444       | 2       | 1950      | 17000.00 |
| 2445       | 2       | 1950      | 16000.00 |
| 2446       | 2       | 1960      | 14000.00 |
| 2447       | 2       | 1960      | 20000.00 |
| 2448       | 3       | 1950      | 18000.00 |
| 2449       | 3       | 1950      | 13000.00 |
| 2450       | 3       | 1960      | 21000.00 |
| 2451       | 3       | 1960      | 22000.00 |
| 2454       | NULL    | 1950      | 13000.00 |

|      |      |      |          |
|------|------|------|----------|
| 2455 | NULL | 1950 | 14000.00 |
| 2456 | NULL | NULL | 19000.00 |

Тогда результат запроса из [примера 16.1](#) имел бы следующий вид<sup>2</sup>:

| DEPT_NO | EMP_BDATE | MAX_SAL  |
|---------|-----------|----------|
| NULL    | NULL      | 22000.00 |
| NULL    | NULL      | 19000.00 |
| NULL    | NULL      | 14000.00 |
| 1       | NULL      | 19000.00 |
| 2       | NULL      | 20000.00 |
| 3       | NULL      | 22000.00 |
| 1       | NULL      | 17000.00 |
| 1       | 1950      | 16000.00 |
| 1       | 1960      | 19000.00 |
| 2       | 1950      | 17000.00 |
| 2       | 1960      | 20000.00 |
| 3       | 1950      | 18000.00 |
| 3       | 1960      | 22000.00 |
| NULL    | 1950      | 14000.00 |

**Рис. 16.2.** Результат запроса с разделом GROUP BY ROLLUP к таблице с неопределенными значениями столбцов группировки

Очевидно, что, просматривая строки таблицы, показанной на [рис.16.2](#), невозможно установить, в какой из первых трех строк *неопределенное значение* столбцов **DEPT\_NO** и **EMP\_BDATE** означает то, что эта строка является сводной для всего предприятия, а не то, что она является сводной для всех служащих с неизвестными номером отдела и годом рождения или просто для всех служащих с неизвестным номером отдела. Аналогичным образом невозможно понять, какая строка в следующей далее паре строк является сводной для всех служащих отдела номер 1, а не для всех служащих отдела номер 1 с неизвестным годом рождения.

Для того чтобы всегда можно было разобраться в результатах запросов, включающих раздел **GROUP BY ROLLUP**, в язык SQL была введена специальная *агрегатная функция* **GROUPING**. Эта функция применяется к столбцу, входящему в список столбцов раздела **GROUP BY ROLLUP**, и принимает целое значение 1 в тех строках результирующей таблицы, в которых соответствующий столбец имеет значение **NULL** по той причине, что строка является сводной для более обобщенной группы. В противном случае функция **GROUPING** принимает значение 0.

Уточним формулировку запроса из [примера 16.1](#) ([пример 16.1a](#)):

```
SELECT DEPT_NO, EMP_BDATE, MAX (EMP_SAL) AS MAX_SAL,
```

```

GROUPING (DEPT_NO) AS GDN, GROUPING (EMP_BDATE)
AS GEB
FROM EMP
GROUP BY ROLLUP (DEPT_NO, EMP_BDATE);
Пример 16.1a.

```

Результирующая таблица для этого запроса будет иметь следующий вид:

| DEPT_NO | EMP_BDATE | MAX_SAL  | GDN | GEB |
|---------|-----------|----------|-----|-----|
| NULL    | NULL      | 22000.00 | 1   | 1   |
| NULL    | NULL      | 19000.00 | 0   | 0   |
| NULL    | NULL      | 14000.00 | 1   | 0   |
| 1       | NULL      | 19000.00 | 0   | 1   |
| 2       | NULL      | 20000.00 | 0   | 1   |
| 3       | NULL      | 22000.00 | 0   | 1   |
| 1       | NULL      | 17000.00 | 0   | 0   |
| 1       | 1950      | 16000.00 | 0   | 0   |
| 1       | 1960      | 19000.00 | 0   | 0   |
| 2       | 1950      | 17000.00 | 0   | 0   |
| 2       | 1960      | 20000.00 | 0   | 0   |
| 3       | 1950      | 18000.00 | 0   | 0   |
| 3       | 1960      | 22000.00 | 0   | 0   |
| NULL    | 1950      | 14000.00 | 0   | 0   |

**Рис. 16.3.** Результат запроса с разделом GROUP BY ROLLUP и вызовами агрегатной функции GROUPING к таблице с неопределенными значениями столбцов группировки

Анализируя значения столбцов **GDN** и **GEB** в строках таблицы, показанной на [рис.16.4](#), можно убедиться, что значение столбца **MAX\_SAL** в первой строке является максимальным значением зарплаты всех служащих предприятия, во второй строке - максимальным значением зарплаты служащих с неизвестными номером отдела и годом рождения, а в третьей строке - максимальным значением зарплаты всех служащих с неизвестным номером отдела. В следующих трех строках значения столбца **MAX\_SAL** являются максимальными значениями зарплаты служащих с неизвестным годом рождения из отделов с номерами 1, 2 и 3 соответственно. Как видно, значения столбцов **GDN** и **GEB** являются своего рода индикаторами, указывающими на природу основных значений строки.

### Раздел GROUP BY CUBE

Наконец, заметим, что, в отличие от запросов с традиционной группировкой, результат запроса, содержащего раздел **GROUP BY ROLLUP**, зависит от порядка столбцов в списке группировки. При выполнении запроса происходит движение по этому списку слева направо с повышением уровня детальности результирующих данных. Существует еще одна разновидность запроса с группировкой, основанная на использовании раздела **GROUP BY CUBE**.

Пусть раздел группировки запроса имеет вид **GROUP BY CUBE** ( cname1, cname2, ... , cnamen ), где cname<sub>i</sub> ( i = 1, 2, ... , n ) - имя столбца таблицы-результата раздела **FROM** запроса. Обозначим через **SGBC** множество {cname<sub>1</sub>, cname<sub>2</sub>, ... , cname<sub>n</sub>}. Пусть **S<sub>i</sub>** является произвольным подмножеством **SGBC**, т.е. **S<sub>i</sub>** представляет собой пустое множество или имеет вид {cname<sub>i1</sub>, cname<sub>i2</sub>, ... , cname<sub>im</sub>}, где m ≤ n, и каждое имя столбца cname<sub>ij</sub> совпадает с одним и только одним именем столбца из списка столбцов раздела **GROUP BY CUBE**. Очевидно, что у множества **SGBC** существует 2<sup>n</sup> подмножеств различных вида **S<sub>i</sub>**. Тогда по определению результат этого запроса совпадает с объединением результатов 2<sup>n</sup> запросов с теми же разделами **SELECT**, **FROM** и **WHERE**, что и у запроса с **GROUP BY CUBE**, и с разделом группировки вида **GROUP BY S<sub>i</sub>**, причем во всех строках результата частичного запроса значением любого столбца cname<sub>j</sub> такого, что cname<sub>j</sub> ∈ **SGBC** и cname<sub>j</sub> ∉ **S<sub>i</sub>**, является **NULL**. Запрос с разделом группировки вида **GROUP BY S**, где **S** - пустое множество, трактуется как запрос без раздела **GROUP BY**. Вот пример запроса, содержащего раздел **GROUP BY CUBE**.

```
SELECT DEPT_NO, EMP_BDATE, MAX (EMP_SAL)AS MAX_SAL,
GROUPING (DEPT_NO) AS GDN, GROUPING (EMP_BDATE) AS GEB
FROM EMP
GROUP BY CUBE (DEPT_NO, EMP_BDATE);
```

Пример 16.2. Найти максимальный размер зарплаты во всем предприятии, максимальный размер зарплаты в каждом отделе, максимальный размер зарплаты служащих в каждой возрастной категории и максимальный размер зарплаты служащих каждой возрастной категории каждого отдела.

Результирующая таблица для этого запроса будет иметь следующий вид:

| DEPT_NO | EMP_BDATE | MAX_SAL  | GDN | GEB |
|---------|-----------|----------|-----|-----|
| NULL    | NULL      | 22000.00 | 1   | 1   |
| NULL    | NULL      | 19000.00 | 0   | 0   |
| NULL    | NULL      | 14000.00 | 1   | 0   |
| 1       | NULL      | 19000.00 | 0   | 1   |
| 2       | NULL      | 20000.00 | 0   | 1   |
| 3       | NULL      | 22000.00 | 0   | 1   |
| 1       | NULL      | 17000.00 | 0   | 0   |
| 1       | 1950      | 16000.00 | 0   | 0   |
| 1       | 1960      | 19000.00 | 0   | 0   |
| 2       | 1950      | 17000.00 | 0   | 0   |
| 2       | 1960      | 20000.00 | 0   | 0   |
| 3       | 1950      | 18000.00 | 0   | 0   |
| 3       | 1960      | 22000.00 | 0   | 0   |
| NULL    | 1950      | 14000.00 | 0   | 0   |
| NULL    | 1950      | 18000.00 | 1   | 0   |
| NULL    | 1960      | 22000.00 | 1   | 0   |

**Рис. 16.4.** Результат запроса с разделом **GROUP BY CUBE** и вызовами агрегатной функции **GROUPING** к таблице с неопределенными значениями столбцов группировки

Как видно, результат запроса из [примера 16.2](#) совсем немного отличается от результата запроса из [примера 16.1a](#). Добавились две последние строки, показывающие максимальные значения зарплаты всех служащих предприятия, родившихся в 1950-м и 1960-м гг. соответственно.

Наш пример может навести на мысль, что и в общем случае запросы, содержащие раздел **GROUP BY CUBE**, не слишком отличаются от запросов с **GROUP BY ROLLUP**, и выполнение этих запросов тоже не слишком различается. Однако это совсем не так. Запрос, содержащий раздел **GROUP BY CUBE**, действительно вырождается в объединение результатов  $2^n$  запросов с обычным разделом **GROUP BY**. Соответственно, сложность выполнения такого запроса несравненно выше сложности выполнения похожего запроса с **GROUP BY ROLLUP**. В нашем примере все получилось так просто только по той причине, что в запросе имеются всего два столбца группировки.

## Рекурсивные запросы

Начнем этот раздел с нескольких определений, касающихся понятий, которые связаны с рекурсией. Эти понятия имеют общий характер, но в приведенных ниже определениях и комментариях к ним (там, где это уместно) подчеркивается *контекст SQL*.

### Определения, относящиеся к рекурсии

**Обход дерева в ширину.** При этом *способе обхода* непосредственные потомки обходятся слева направо, до того как производится переход к потомкам следующего уровня родства.

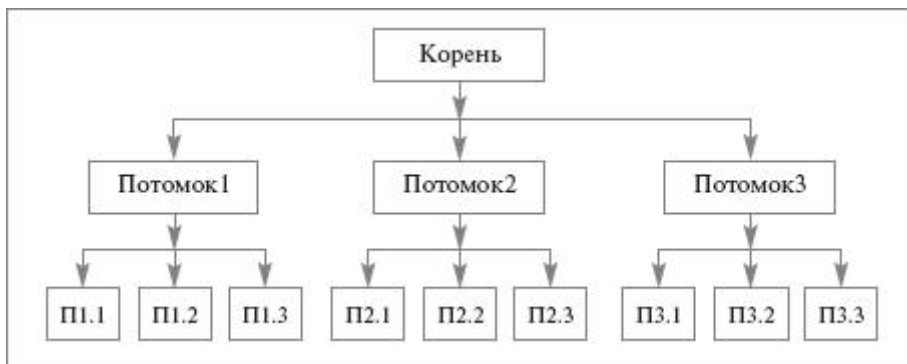


**Рис. 16.5.** Пример дерева

При *обходе в ширину* дерева, показанного на [рис.16.5](#), узлы будут обходиться в следующем порядке: **Корень-Потомок1-Потомок2-Потомок3-П1.1-П1.2-П1.3-П2.1-П2.2-П2.3-П3.1-П3.2-П3.3**.

**Обход дерева в глубину.** При этом *способе обхода* на каждом шаге производится переход к самому левому текущему потомку. При *обходе в глубину* дерева с [рис.16.5](#) порядок обхода узлов будет следующим: **Корень-Потомок1-П1.1-П1.2-П1.3-Потомок2-П2.1-П2.2-П2.3-Потомок3-П3.1-П3.2-П3.3**.

**Цикл в ориентированном графе.** В теории графов *ориентированный граф* называется циклическим в том и только в том случае, когда хотя бы один узел графа одновременно является и предком, и потомком (т. е. для этого узла имеется и выходящая, и входящая дуги). В SQL:1999 узлами графа рекурсии являются строки, входящие в результат *рекурсивного запроса*, а дуги соответствуют способам обработки текущих строк,

которые ведут к добавлению к результату новых строк. На [рис.16.6](#) показан простейший пример *ориентированного графа с циклом*.



Рис. 16.6. Пример графа с циклом

**Прямая рекурсия.** По определению, некоторый элемент использует *прямую рекурсию* в том и только в том случае, когда он обращается сам к себе без посредников. Пример, приведенный на [рис.16.6](#), демонстрирует (в графовой форме) *прямую рекурсию*. На [рис.16.7](#) показан графовый пример *непрямой рекурсии*.



Рис. 16.7. Графовый пример непрямой рекурсии

**Линейная рекурсия.** При линейно рекурсивном вызове элемент прямо рекурсивно обращается сам к себе не более одного раза. В SQL:1999 в определении любой *виртуальной таблицы* с рекурсией допускается не более одной ссылки на саму себя (в разделе **FROM** и/или в *подзапросах*). На [рис.16.8](#) показан графовый пример рекурсии, не являющейся *линейной*.

Монотонность. **Монотонной прогрессией** называется последовательность неубывающих или невозрастающих значений. Например, последовательность натуральных чисел  $\{1, 2, \dots, n, \dots\}$  является монотонной. В SQL:1999 свойство монотонности поддерживается в том смысле, что число строк результата *рекурсивного запроса* не уменьшается на каждом шаге рекурсии.

**Взаимная рекурсия.** Элементы **A** и **B** связаны отношением *взаимной рекурсии*, если **A** прямо или косвенно вызывает **B**, и **B** прямо или косвенно вызывает **A**. На [рис.16.9](#) показан графовый пример *взаимной рекурсии* (элемент **A** вызывает элемент **B** через элемент **C**, а элемент **B** вызывает элемент **A** через элемент **D**).



Рис. 16.8. Графовый пример нелинейной рекурсии

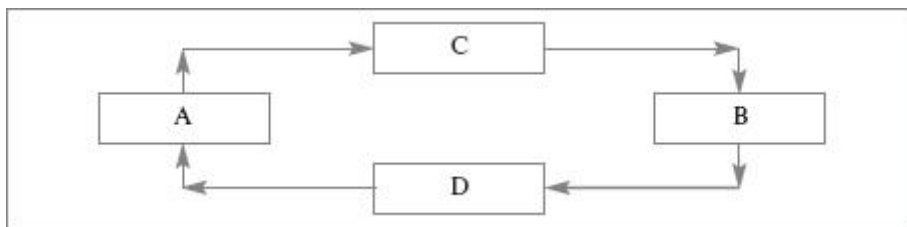


**Рис. 16.9.** Графовый пример взаимной рекурсии

**Отрицание.** В контексте SQL *отрицанием* называется любое действие, приводящее к уменьшению числа строк в результате запроса. Свойствами *отрицания* обладают операции над (мульти)множествами **EXCEPT** и **INTERSECT**, спецификация **DISTINCT**, условие **NOT EXISTS** и т.д. В стандарте SQL не запрещается использование *отрицания* в *рекурсивных запросах*. Возможной проблемы нарушения монотонности удастся избежать за счет того, что *отрицание* разрешается применять только к тем таблицам, которые являются полностью известными (или вычисленными) к моменту применения *отрицания*. В процессе вычисления таблицы применение к ней *отрицания* не допускается.

**Начальный источник рекурсии.** При выполнении рекурсивных вычислений обычно (хотя и не всегда) имеется некоторое начальное значение. В SQL этим *начальным источником рекурсии* является одна или несколько строк, удовлетворяющих некоторым начальным условиям. На основе этих строк в процессе *рекурсивного вычисления* производятся дополнительные строки, образующие окончательный результат.

**Стратификация.** В SQL *рекурсивный запрос* обычно состоит из "рекурсивной" и "нерекурсивной" частей. В процессе *стратификации* ("расслоения") запроса выполнение этих двух частей разделяется. В более сложных *рекурсивных запросах* может содержаться несколько рекурсивных частей и более одной нерекурсивной части. В этом случае в процессе *стратификации* будет обнаружено большее число слоев.

**Семантика фиксированной точки.** В контексте SQL:1999 *семантика фиксированной точки* означает, что решение о завершении *рекурсивного запроса* принимается тогда, когда становится невозможно добавить к результату какие-либо дополнительные строки.

## Рекурсивные запросы с разделом WITH

В предыдущих лекциях мы уже говорили о разновидности спецификации ссылки на таблицу с использованием раздела **WITH**. Однако мы умышленно отложили обсуждение рекурсивных возможностей. Полный синтаксис раздела **WITH** выглядит следующим образом:

```

with_clause ::= WITH [RECURSIVE]
with_element_comma_list
with_element ::= query_name [(column_name_list)]
 AS (query_expression) [search_or_cycle_clause]
search_or_cycle_clause ::= search_clause
 | cycle_clause
 | search_clause cycle_clause
search_clause ::= SEARCH recursive_search_order SET
 sequence_column_name

```

```

recursive_search_order ::= DEPTH FIRST BY
 order_item_commalist
 | BREADTH FIRST BY order_item_commalist
cycle_clause ::= CYCLE cycle_column_name_comma_list
 SET cycle_mark_column_name TO value_expression
 DEFAULT value_expression
 USING path_column_name

```

Для иллюстрации возможностей *рекурсивных запросов* с разделом **WITH** и пояснения смысла конструкций **SEARCH** и **CYCLE** воспользуемся классическим примером "разборки деталей" (в данном случае мы будем разбирать автомобиль). Предположим, что данные о конструктивных элементах автомобиля хранятся в таблице **CAR**, определенной следующим образом:

```

CREATE TABLE CAR (CONTAINING_PART VARCHAR (10),
 CONTAINED_PART VARCHAR (10),
 NUMBER_OF_PARTS INTEGER,
 PART_COST DECIMAL (6,2));

```

У автомобиля имеется один конструктивный элемент верхнего уровня - полностью собранный автомобиль. Этот элемент не является составной частью какого-либо другого элемента, и для его строки значением столбца **CONTAINING\_PART** является текстовая строка длины 0. В любой другой строке таблицы **CAR**, соответствующей некоторому неатомарному конструктивному элементу **e**, столбец **CONTAINING\_PART** содержит идентификационный номер элемента **e1**, в который входит элемент **e**, столбец **NUMBER\_OF\_PARTS** - число экземпляров элемента **e**, входящих в **e1**, а столбец **CONTAINED\_PART** - идентификационный номер самого элемента **e**. В любой строке таблицы **CAR**, соответствующей некоторому *атомарному* конструктивному элементу, значением столбца **CONTAINED\_PART** является строка длины 0, а в столбце **PART\_COST** сохраняется цена атомарного конструктивного элемента (для неатомарных элементов значение этого столбца равно нулю).

Предположим, что нам требуется разобрать автомобиль, начиная с элемента самого верхнего уровня, и для каждого конструктивного элемента получить его номер, общее число используемых экземпляров этого элемента, а также, если элемент является атомарным, общую стоимость используемых экземпляров. Вот возможная формулировка запроса ([пример 16.3](#)):

```

WITH RECURSIVE PARTS (PART_NUMBER,
 NUMBER_OF_PARTS, COST) AS
 (SELECT CONTAINED_PART, 1, 0.00 (a)
 FROM CAR
 WHERE CONTAINING_PART = ''
 UNION ALL
 SELECT CAR.CONTAINED_PART, CAR.NUMBER_OF_PARTS,
 CAR.NUMBER_OF_PARTS * CAR.PART_COST
 FROM CAR, PARTS
 WHERE PARTS.PART_NUMBER = CAR.CONTAINING_PART)
SELECT PART_NUMBER, SUM(NUMBER_OF PARTS),

```

```
SUM(COST) (b)
FROM PARTS
GROUP BY PART_NUMBER;
Пример 16.3.
```

Этот запрос будет выполняться следующим образом. При вычислении раздела **FROM** основного запроса ( **b** ) начнется выполнение *рекурсивного* выражения запросов ( **a** ), определенного в разделе **WITH** . На первом шаге рекурсии будет выполнена часть данного выражения, предшествующая операции **UNION ALL** и образующая *начальный источник рекурсии*. В результате будет произведено исходное состояние *виртуальной таблицы PARTS*, в котором, в нашем случае, появится единственная строка, соответствующая автомобилю целиком. На следующем шаге к таблице **PARTS** будут добавлены строки, соответствующие конструктивным элементам второго уровня (для автомобиля это, по-видимому, двигатель, колеса, *шасси* и т.д.). Этот процесс будет продолжаться до тех пор, пока мы не дойдем до атомарных конструктивных элементов и не достигнем, тем самым, *фиксированной точки*. Поскольку в *рекурсивном запросе* содержится операция **UNION ALL**, в результирующей таблице могут появляться строки-дубликаты. Наличие строки-дубликата вида **<part\_no, number, cost>** означает, что элемент с номером **part\_no** входит в одном и том же числе экземпляров в несколько конструктивных элементов более высокого уровня.

## Раздел SEARCH

В приведенном выше примере не определялся порядок, в котором строки добавляются к *частичному результату рекурсивного запроса*. Однако иногда требуется, чтобы иерархия обходилась *в глубину* или *в ширину*. Соответствующая возможность обеспечивается конструкцией **SEARCH** . При указании требования *обхода в глубину* гарантируется, что каждый элемент-предок появится в результате раньше своих потомков и своих братьев справа. Если указывается требование обхода иерархии *в ширину*, в результате все братья одного уровня появляются раньше, чем какой-либо их *потомок*. Ниже показан вариант запроса, в котором содержится раздел **SEARCH** с требованием обхода иерархии элементов автомобиля *в ширину* ([пример 16.4](#)).

```
WITH RECURSIVE PARTS (ASSEMBLY, PART_NUMBER,
 NUMBER_OF_PARTS, COST) AS
(SELECT CONTAINING_PART, CONTAINED_PART, 1, 0.00
FROM CAR
WHERE CONTAINING_PART = "
UNION ALL
SELECT CAR.CONTAINING_PART, CAR.CONTAINED_PART,
 CAR.NUMBER_OF_PARTS, CAR.NUMBER_OF_PARTS *
 CAR.PART_COST
FROM CAR, PARTS
WHERE PARTS.PART_NUMBER = CAR.CONTAINING_PART)
SEARCH BREADTH FIRST
BY CONTAINING_PART, CONTAINED_PART
SET ORDER_COLUMN
SELECT PART_NUMBER, NUMBER_OF PARTS, COST
FROM PARTS
ORDER BY ORDER_COLUMN;
```

#### Пример 16.4.

В списке столбцов сортировки раздела **SEARCH** должны указываться имена столбцов *виртуальной таблицы*, определенной в разделе **WITH**. Поскольку в данном случае мы хотим, чтобы в результате сначала появлялись все конструктивные элементы одного уровня (**CONTAINING\_PART**), а затем все их подэлементы (**CONTAINED\_PART**), в список выборки *рекурсивного запроса* **PARTS** добавлен столбец **CONTAINING\_PART**, который не используется нигде, кроме раздела **SEARCH**. В разделе **SET** к результирующей таблице *рекурсивного запроса* добавлен столбец, который мы назвали **ORDER\_COLUMN**. Название соответствует природе столбца, потому что при выполнении *рекурсивного запроса* в этот столбец автоматически заносятся значения, характеризующие порядок генерируемых строк в соответствии с выбранным способом обхода иерархии. Чтобы строки результата основного запроса появлялись в должном порядке, в этом запросе требуется наличие раздела **ORDER BY** с указанием столбца, определенного в разделе **SET**.

#### Раздел CYRCLE

Наконец, обсудим, для чего нужен раздел **CYRCLE**. Дело в том, что иногда сами данные, хранимые в таблицах базы данных, могут иметь циклическую природу. Представим себе, например, компанию, в которой существует совет директоров, являющийся высшим органом управления компанией. Обычным случаем является тот, когда по крайней мере один из членов совета директоров является простым служащим этой же компании (например, он может входить в совет директоров как представитель профсоюза). Назовем данного члена совета директоров **EMP\_DIR**. Как член совета директоров, **EMP\_DIR** "управляет" деятельностью президента компании. С другой стороны, как служащий компании, **EMP\_DIR** находится в прямом или косвенном подчинении у президента компании. Такое положение может привести к заикливанию выполнения *рекурсивных запросов*. Раздел **CYRCLE** обеспечивает некоторую возможность распознавать подобные ситуации. Если у пользователя имеется полная уверенность в отсутствии циклов в данных, к которым адресуется *рекурсивный запрос*, то использование раздела **CYRCLE** не требуется.

Подход к распознаванию заикленных запросов, принятый в SQL, состоит в том, что распознаются данные, которые уже участвовали ранее в формировании результата *рекурсивного запроса*. При наличии раздела **CYRCLE** при добавлении к результату строк, удовлетворяющих условию запроса, такие строки помечаются указанным значением, которое означает, что эти строки уже вошли в результат. При попытке добавления к результату каждой новой строки проверяется, не находится ли она уже в результате, т. е. не помечена ли она этим указанным в разделе **CYRCLE** значением. Если это действительно так, то считается, что имеет место цикл, и дальнейшее выполнение *рекурсивного запроса* прекращается.

Обсудим все это более формально. Для удобства воспроизведем еще раз синтаксис раздела **CYRCLE**.

```
cycle_clause ::= CYCLE cycle_column_name_comma_list
 SET cycle_mark_column_name TO value_expression_1
 DEFAULT value_expression_2
 USING path_column_name
```

В списке `cycle_column_name_comma_list` указываются имена одного или нескольких столбцов, которые используются для идентификации новых строк результата на основе строк, уже входящих в результат. Например, в примерах [16.3](#) и [16.4](#) столбец `CONTAINED_PART` связывает конструктивный элемент автомобиля с входящими в его состав подэлементами (через значения их столбцов `CONTAINING_PART`). Раздел `SET` приводит к образованию нового столбца результирующей таблицы. Для строк, которые попадают в результат первый раз, в столбец `cycle_mark_column_name` заносится значение выражения `value_expression_2`. В повторно заносимых строках значение столбца - `value_expression_1`. Типом данных этого столбца является *тип символьных строк* длины один, так что в качестве `value_expression_1` и `value_expression_2` разумно использовать константы `'0'` и `'1'` или `'Y'` и `'N'`.

Раздел `USING` приводит к образованию еще одного дополнительного столбца результата с именем `path_column_name`. Типом данных столбца является `ARRAY`, причем кардинальность этого типа предполагается достаточно большой, чтобы сохранить информацию обо всех строках, попавших в результат. Элементы массива имеют "строчный тип" (`row type`), содержащий столько столбцов, сколько их указано в списке раздела `CYRCLE`. Каждый элемент массива соответствует строке результата, и в его столбцах содержится копия значений соответствующих столбцов этой строки. Вот пример запроса, содержащего раздел `CYRCLE` ([пример 16.5](#)):

```
WITH RECURSIVE PARTS (PART_NUMBER,
 NUMBER_OF_PARTS, COST) AS
(SELECT CONTAINED_PART, 1, 0.00
 FROM CAR
 WHERE CONTAINING_PART = "
UNION ALL
 SELECT CAR.CONTAINED_PART, CAR.NUMBER_OF_PARTS,
 CAR.NUMBER_OF_PARTS * CAR.PART_COST
 FROM CAR, PARTS
 WHERE PARTS.PART_NUMBER = CAR.CONTAINING_PART)
CYRCLE CONTAINED_PART
SET CYCLEMARK TO 'Y' DEFAULT 'N'
USING CYRCLEPATH
SELECT PART_NUMBER, SUM(NUMBER_OF PARTS), SUM(COST)
FROM PARTS
ORDER BY PART_NUMBER;
```

Пример 16.5.

Имена столбцов `CYCLEMARK` и `CYRCLEPATH` выбраны произвольным образом - требуется только, чтобы имена этих столбцов отличались от имен столбцов *рекурсивного запроса*. При выполнении запроса строки, удовлетворяющие его условию, накапливаются в результирующей таблице. Но, кроме того, эти строки "кэшируются" в столбце `CYRCLEPATH`. При попытке добавления к результату новой строки на основе текущего содержимого столбца `CYRCLEPATH` проверяется, не содержится ли она уже в результате. Если не содержится, то данные об этой строке добавляются к столбцу `CYRCLEPATH` (к массиву добавляется новый элемент), в столбец `CYCLEMARK` этой строки заносится значение `'N'`, и строка добавляется к

результату. Иначе в столбец **CYCLEMARK** соответствующей строки результата заносится значение 'Y', означающее, что от этой строки начинается цикл.

## Рекурсивные представления

Рекурсивным называется *представление*, в определяющем выражении запроса которого используется имя этого же *представления*. В *представлениях* может использоваться и *прямая*, и *взаимная рекурсия*. Синтаксис оператора определения *рекурсивного запроса* выглядит следующим образом:

```
CREATE RECURSIVE VIEW table_name
[column_name_comma_list]
AS query_expression
```

Хотя для того, чтобы *представление* было *рекурсивным*, требуется рекурсивность определяющего выражения *запроса* (т.е. в нем должна присутствовать спецификация **RECURSIVE**); наличие избыточного ключевого **RECURSIVE** в определении *рекурсивного представления* является обязательным. Как говорят авторы стандарта, это сделано для того, чтобы избежать случайного появления непредусмотренных *рекурсивных представлений*. Наконец, обратите внимание на то, что еще не обсуждавшийся нами необязательный раздел **WITH CHECK OPTION** не может присутствовать в определении *рекурсивного представления* (по той причине, что разработчики стандарта не смогли найти разумной интерпретации для комбинации **RECURSIVE** и **WITH CHECK OPTION**).

В заключение этого раздела могу сказать, что лично мне механизм рекурсии, предлагаемый в стандарте SQL, представляется громоздким и ограниченным. Кроме того, насколько мне известно, компании, поставляющие SQL-ориентированные СУБД, не спешат внедрять в свои продукты средства рекурсии в соответствии со стандартом SQL:1999 (или, по крайней мере, не слишком их афишируют).

## Заключение

Если вернуться к синтаксическим определениям разд. "Общие синтаксические правила построения скалярных выражений" лекции 13, то можно убедиться, что в последних четырех лекциях мы рассмотрели все варианты организации оператора **SELECT** языка SQL (за исключением конструкций **collection\_derived\_table** и **ONLY ( table\_or\_query\_name )**), относящихся к объектным расширениям языка SQL).

Для общего понимания языка на модельном уровне более важными являются предыдущие три лекции. Данная Лабораторная работа включена в курс, скорее, с целью общего ознакомления читателей с новыми возможностями оператора выборки, чем с целью их подробного описания. С большой вероятностью средства формулировки аналитических и *рекурсивных запросов* языка SQL будут пересматриваться при подготовке следующих вариантов *стандарта языка*.

## Лабораторная работа 7. Средства манипулирования данными

### Введение

*Базы данных*, по крайней мере, в приложениях категории *OLTP*, являются высоко динамичными объектами. В таких приложениях на две *операции* выборки данных в среднем приходится одна операция обновления содержимого *базы данных* (добавления новых данных, удаления или модификации существующих данных). Поэтому для пользователей и разработчиков *OLTP*-приложений средства **манипулирования данными** по важности находятся на втором месте после средств выборки данных.

В этой лекции мы обсудим средства манипулирования данными, входящие в **прямой SQL**. Заметим, что с практической точки зрения более важными являются средства манипулирования данными, выходящие за пределы прямого **SQL** и присутствующие во **встраиваемом** и **динамическом SQL**. Но, как мы неоднократно отмечали, в этом курсе мы не обсуждаем возможности использования **SQL** для создания приложений. Помнению автора, материал данной лекции полезен для общего понимания специфики операторов манипулирования данными, а расширения этих операторов, присутствующие во **встраиваемом** и **динамическом SQL**, в любом случае нужно изучать совместно с другими аспектами подобных уровней языка.

Лабораторная работасостоит из трех основных разделов. В первом разделе мы обсудим *синтаксис* и семантику операторов манипулирования данными, полагая, что они действуют над *базовыми таблицами*. Во втором разделе будет продемонстрировано, что в ряде случаев, специфицированных в *стандарте языка SQL*, *операторы* манипулирования данными можно применять к порождаемым таблицам и *представлениям* с однозначным отображением результатов действия этих операторов на соответствующие *базовые таблицы*. Третий раздел посвящен механизму *триггеров*, которые, по существу, представляют собой "хранимые процедуры", автоматически вызываемые при возникновении соответствующих условий. *Триггеры* не обязательно связываются с действиями, производимыми при манипулировании данными, но, поскольку одно из основных функций этого механизма состоит в поддержании целостности баз данных, как правило, такая *связь* имеется. Поэтому мы включили обсуждение механизма *триггеров* в соответствии со стандартом **SQL** именно в данную лекцию.

## Базовые средства манипулирования данными

К базовым средствам манипулирования данными языка **SQL** относятся "поисковые" варианты операторов **UPDATE** и **DELETE**. Эти варианты называются поисковыми, потому что при задании соответствующей *операции* задается логическое условие, налагаемое на строки адресуемой оператором таблицы, которые должны быть подвергнуты модификации или удалению. Кроме того, в такую категорию языковых средств входит оператор **INSERT**, позволяющий добавлять строки в существующие таблицы. Логично начать изложение именно с оператора **INSERT**, поскольку, для того чтобы можно было что-либо модифицировать в таблицах или удалять из таблиц, нужно, чтобы в таблицах содержались какие-то строки.

### Оператор INSERT для вставки строк в существующие таблицы

Общий синтаксис оператора **INSERT** выглядит следующим образом:

```
INSERT INTO table_name
 { [(column_commalist)] query_expression
 | DEFAULT VALUES
```

На вид синтаксические правила кажутся очень простыми, пока не вспомнишь, что обозначает синтаксическая категория **query\_expression** (см. раздел "Общие синтаксические правила построения *скалярных выражений*" лекции 13). Даже если ограничиться простейшей составляющей этой конструкции ( **simple\_table** ), то мы имеем следующие возможности:

```
simple_table ::= query_specification
 | table_value_constructor
 | TABLE table_name
```

#### **Вставка всех строк указанной таблицы**

Тем самым, стандарт допускает *вставку в указанную таблицу всех строк некоторой другой таблицы* (вариант **table\_name** ). Эта другая таблица может быть как базовой, так и представляемой. Естественно, что в последнем случае в определении *представления* не должны присутствовать ссылки на таблицу, в которую производится вставка. При использовании данного варианта оператора вставки число столбцов вставляемой таблицы должно совпадать с числом столбцов таблицы, в которую производится вставка, или с числом столбцов, указанных в списке **column\_commlist**, если этот *список задан*. Типы данных соответствующих столбцов вставляемой таблицы и таблицы, в которую производится вставка, должны быть совместимыми. Если в операции задан список **column\_commlist** и в нем содержатся не все имена столбцов таблицы, в которую производится вставка, то в оставшиеся столбцы во всех строках заносятся значения столбцов по умолчанию. Если для какого-либо из оставшихся столбцов значение по умолчанию не определено, при выполнении операции вставки фиксируется ошибка.

Чтобы привести пример этого варианта операции *INSERT* (**пример 17.1**), предположим, что в базе данных *EMP-DEPT-PRO* имеется еще одна промежуточная таблица *EMP\_TEMP*, в которой временно хранятся данные о служащих, проходящих испытательный срок. Пусть эта таблица имеет следующий заголовок:

```
EMP_TEMP:

EMP_NO : EMP_NO

EMP_NAME : VARCHAR

EMP_BDATE : DATE
```

В таблице **EMP\_TEMP** хранятся не полные сведения о служащих, а именно те, которые требуются на время испытательного срока. Если выполнить операцию

```
INSERT INTO EMP (EMP_NO, EMP_NAME, EMP_BDATE) TABLE EMP_TEMP;
```

**Пример 17.1.**

то в основной таблице **EMP** появятся строки, соответствующие служащим, проходившим испытательный срок. При этом в столбцах **EMP\_NO**, **EMP\_NAME**, **EMP\_BDATE** этих

строк будут содержаться данные, взятые из таблицы **EMP\_TEMP**, а в столбцах **EMP\_SAL**, **DEPT\_NO**, **PRO\_NO** будут находиться значения, определенные для данных столбцов по умолчанию. Конечно, поскольку столбец **EMP\_NO** является *первичным ключом* таблицы **EMP** (по всей видимости, и таблицы **EMP\_TEMP**), операция вставки будет успешно выполнена только в том случае, когда *ограничение первичного ключа* таблицы **EMP** не будет нарушено (конечно же, требуется выполнение и всех других *ограничений целостности*, определенных для таблицы **EMP**).

### Вставка явно заданного набора строк

Теперь обратимся к варианту оператора **INSERT**, в котором *набор вставляемых строк задается явно* с использованием синтаксической конструкции **table\_value\_constructor**. Напомним синтаксические правила, определяющие эту конструкцию:

```
table_value_constructor ::=
 VALUES row_value_constructor_comma_list
row_value_constructor ::= row_value_constructor_element
 | [ROW] (row_value_constructor_element_comma_list)
 | row_subquery
row_value_constructor_element ::= value_expression
 | NULL | DEFAULT
```

Самый простой пример использования этого варианта оператора вставки состоит в занесении в таблицу **EMP** явно задаваемых данных о новом служащем ([пример 17.2](#)):

#### INSERT INTO EMP

```
ROW (2445, 'Brown', '1985-04-08', 16500.00, 630, 772);
```

Пример 17.2.

В этом примере явно заданы значения всех столбцов заносимой строки (как показывают синтаксические правила, ключевое слово **ROW** можно опустить). Возможен и такой вариант ([пример 17.2a](#)):

#### INSERT INTO EMP

```
ROW (2445, DEFAULT, NULL, DEFAULT, NULL, NULL);
```

Пример 17.2a.

В этом случае мы знаем о новом служащем очень мало, но уверены в том, что его имя и размер *заработной платы* должны быть назначены по умолчанию, а про дату рождения, номер отдела и номер проекта ничего не известно. Обратите внимание, что выполнение подобной операции не нарушает *ограничения целостности* таблицы **EMP**.

Если обладать полной информацией об определении таблицы **EMP**, то формулировку операции [примера 17.2a](#) можно переписать короче следующим эквивалентным образом ([пример 17.2b](#)):

```
INSERT INTO EMP (EMP_NO) 2445;
```

Пример 17.2b.

Вспомним теперь, что одной из разновидностей **value\_expression\_primary** является **scalar\_subquery** (см. раздел "Скалярные выражения" лекции 13). Это означает, что в список элементов конструктора строки могут входить скалярные запросы, т. е. запросы, результат выполнения которых состоит из единственной строки, включающей единственный столбец. Поэтому допустима, например, такая операция вставки ([пример 17.3](#)):

```
INSERT INTO EMP VALUES
 ROW (2445, (SELECT EMP_NAME
 FROM EMP
 WHERE EMP_NO = 2555),
 '1985-04-08',
 (SELECT EMP_SAL
 FROM EMP
 WHERE EMP_NO = 2555),
 NULL, NULL),
 ROW (2446, (SELECT EMP_NAME
 FROM EMP
 WHERE EMP_NO = 2556),
 '1978-05-09',
 (SELECT EMP_SAL
 FROM EMP
 WHERE EMP_NO = 2556),
 NULL, NULL);
```

**Пример 17.3.**

После выполнения этой операции в таблице **EMP** появятся две новые строки для служащих с уникальными идентификаторами **2445** и **2446**, причем первому из них будет присвоено имя и размер *заработной платы* служащего с уникальным идентификатором **2555**, а второму - аналогичные данные о служащем с уникальным идентификатором **2556**.

### **Вставка строк результата запроса**

Наконец, обсудим вариант оператора *вставки*, когда набор *вставляемых строк* определяется через *спецификацию запроса*. Предположим, например, что требуется сохранить в отдельной таблице **DEPT\_SUMMARY** сведения о числе служащих каждого отдела, их максимальной, минимальной и суммарной *заработной плате*. Пусть таблица **DEPT\_SUMMARY** уже создана и имеет следующий заголовок<sup>1</sup>:

DEPT\_SUMMARY:

DEPT\_NO : DEPT\_NO

DEPT\_EMP\_NO : INTEGER

```
DEPT_MAX_SAL : SALARY

DEPT_MIN_SAL : SALARY

DEPT_TOTAL_SAL : SALARY
```

Тогда заполнить таблицу можно с помощью следующей операции вставки (пример 17.4):

```
INSERT INTO DEPT_SUMMARY
(SELECT DEPT_NO, COUNT(*), MAX (EMP_SAL),
 MIN (EMP_SAL), SUM (EMP_SAL)
FROM EMP
GROUP BY DEPT_NO);
```

Пример 17.4.

**Оператор UPDATE для модификации существующих строк в существующих таблицах**

Общий синтаксис оператора **UPDATE** выглядит следующим образом:

```
UPDATE table_name SET update_assignment_commalist
 WHERE conditional_expression
update_assignment ::= column_name =
 { value_expression | DEFAULT | NULL }
```

Семантика оператора модификации существующих строк определяется следующим образом:

1. для всех строк таблицы с именем **table\_name** вычисляется *булевское выражение* **conditional\_expression**. Строки, для которых значением этого *булевского выражения* является **true**, считаются подлежащими модификации (обозначим множество таких строк через  $T_m$ );
2. каждая строка **s** ( $s \in T_m$ ) подвергается модификации таким образом, что значение каждого столбца этой строки, указанного в списке **update\_assignment\_commalist**, заменяется значением, указанным в правой части соответствующего элемента списка модификации<sup>2</sup>. Значения столбцов строки **s**, не указанные в списке модификации, остаются неизменными.

Приведем примеры операций модификации таблиц.

```
UPDATE EMP SET DEPT_NO = 632, EMP_SAL = EMP_SAL + 1000.00
 WHERE PRO_NO = 772;
```

Пример 17.5. Перевести всех служащих, выполняющих проект с номером 772, в отдел 632 и повысить им заработную плату на 1000 руб.

При выполнении данной операции на первом шаге в таблице **EMP** будут найдены все строки, относящиеся к служащим, которые участвуют в проекте с номером 772. На втором

шаге во всех этих строках значение столбца **DEPT\_NO** будет изменено на **632**, а к значению столбца **EMP\_SAL** будет прибавлено **1000.00**.

```
UPDATE EMP SET EMP_SAL = (SELECT AVG (EMP1_SAL)
FROM EMP EMP1
WHERE EMP.DEPT_NO = EMP1.DEPT_NO)
+ 1000.00, PRO_NO = NULL
WHERE (SELECT EMP1.EMP_SAL
FROM EMP EMP1, DEPT
WHERE EMP.DEPT_NO = DEPT.DEPT_NO
AND DEPT_MNG = EMP1.EMP_NO) > 30000.00;
```

Пример 17.6. Для всех служащих, работающих в отделах, заработная плата менеджеров которых превышает 30000 руб., установить размер заработной платы, на 1000 руб. превышающий средний размер заработной платы соответствующего отдела, а номера проектов, в которых участвуют эти служащие, сделать неопределенными.

Конечно, если вам больше нравится другой стиль, то запрос, фигурирующий в разделе **WHERE**, можно переформулировать с использованием вложенного *подзапроса* (пример 17.6а).

```
UPDATE EMP SET EMP_SAL = (SELECT AVG (EMP1_SAL)
FROM EMP EMP1
WHERE EMP.DEPT_NO = EMP1.DEPT_NO)
+ 1000.00, PRO_NO = NULL
WHERE DEPT.NO IN (SELECT DEPT.DEPT_NO
FROM EMP, DEPT
WHERE DEPT_MNG = EMP_NO
AND EMP_SAL > 30000.00);
```

Пример 17.6а.

Эти примеры позволяют понять, насколько богаты возможности оператора **UPDATE**. В разделе **WHERE** может содержаться любое условие, допускаемое в операторе выборки, а в элементах списка раздела **SET** может присутствовать любой вид **value\_expression**, в том числе любой запрос, вырабатывающий одиночное значение (*скалярный подзапрос*).

## Оператор DELETE для удаления строк в существующих таблицах

Общий синтаксис оператора **DELETE** выглядит следующим образом:

```
DELETE FROM table_name
WHERE conditional_expression
```

В некотором смысле оператор **DELETE** является частным случаем оператора **UPDATE** (или, наоборот, действие оператора **UPDATE** представляет собой комбинацию действий операторов **DELETE** и **INSERT**).

Семантика оператора модификации существующих строк определяется следующим образом:

1. для всех строк таблицы с именем **table\_name** вычисляется *булевское выражение* **conditional\_expression**. Строки, для которых значением этого *булевского выражения* является **true**, считаются подлежащими удалению (обозначим множество таких строк через  $T_d$ );
2. каждая строка  $s$  ( $s \in T_d$ ) удаляется из указанной таблицы.

С целью иллюстрации приведем два примера операции удаления строк.

**DELETE FROM EMP WHERE PRO\_NO = 772;**

Пример 17.7. Удалить из таблицы EMP все строки, относящиеся к служащим, которые участвуют в проекте с номером 772.

**DELETE FROM EMP WHERE EMP\_SAL >**

**(SELECT EMP1.EMP\_SAL  
FROM EMP EMP1, DEPT  
WHERE EMP.DEPT\_NO = DEPT.DEPT\_NO  
AND DEPT.DEPT\_MNG = EMP1.EMP\_NO);**

Пример 17.8. Удалить из таблицы EMP все строки, относящиеся к служащим, размер заработной платы которых превышает размер заработной платы менеджеров их отделов.

Как и в операторе **UPDATE**, в разделе **WHERE** оператора **DELETE** можно использовать любой вид *булевского выражения*, допустимого в операторе выборки. Поэтому возможности оператора удаления строк ограничены лишь фантазией пользователя.

### Представления, над которыми возможны операции обновления

В разделе "Общие синтаксические правила построения *скалярных выражений*" лекции 13 было введено понятие *представления* (**VIEW**). Кратко повторим, что *представление* - это сохраняемое в каталоге базы данных выражение запросов, обладающее собственным именем и, возможно, собственными именами столбцов. Для удобства повторим синтаксические правила определения *представления*:

**create\_view ::= CREATE [ RECURSIVE ] VIEW table\_name  
[ column\_name\_comma\_list ]  
AS query\_expression  
[ WITH [ CASCADED | LOCAL ] CHECK OPTION ]**

В операциях выборки к любому *представлению* можно адресоваться таким же образом, как и к любой *базовой таблице*. Естественно, возникает вопрос: а можно ли использовать имена *представлений* и в операциях обновления *базы данных* и если такая возможность допускается, то как это следует понимать?

Напомним, что в соответствии с семантикой языка SQL при выполнении запроса, в разделе **FROM** которого прямо или косвенно присутствует имя *представления*, прежде всего, производится *материализация представления*, т.е. вычисляется результат соответствующего выражения запросов, сохраняется во временной *базовой таблице*, и далее запрос выполняется по отношению к этой *базовой таблице*. Хотя в реализациях SQL обычно стремятся избегать *материализации представлений*, любая реализация обязана обеспечить такое выполнение запроса над *представлением*, которое было бы эквивалентно выполнению запроса с явной *материализацией представления*.

Если допустить *выполнение над представлениями операций обновления* (сразу заметим, что, вообще говоря, в языке SQL это всегда разрешалось), то в этом случае *семантика материализации* явно не подходит. На первое место выходит требование, чтобы *операция обновления над представлением* однозначно отображалась в одну или несколько операций обновления над теми постоянно хранимыми базовыми таблицами, над которыми прямо или косвенно определено данное представление.

## Представления, допускающие применение операций обновления, в стандарте SQL/92

Поскольку базовым элементом выражения запросов является спецификация запроса, прежде всего нужно понять, какой класс спецификаций запросов является **допускающим операции обновления** (термин **updatable** - **обновляемый**, используемый в стандарте SQL, кажется не слишком удачным в русском варианте). В стандарте SQL/92 спецификация запроса считалась *допускающей операции обновления* в том и только в том случае, когда выполнялись следующие условия:

- в разделе **SELECT** спецификации запроса отсутствует ключевое слово **DISTINCT** (т.е. не требуется удаление строк-дубликатов из результата запроса);
- все элементы списка выборки раздела **SELECT** являются именами столбцов, и ни одно имя столбца не встречается в этом списке более одного раза;
- в разделе **FROM** присутствует только одна ссылка на таблицу, и она указывает либо на базовую таблицу, либо на порождаемую таблицу, допускающую операции обновления;
- прямые или косвенные ссылки на базовую таблицу, прямо или косвенно идентифицируемую ссылкой на таблицу в разделе **FROM**, не встречаются в разделе **FROM** ни одного подзапроса, участвующего в разделе **WHERE** спецификации запроса;
- в спецификации запроса отсутствуют разделы **GROUP BY** и **HAVING**.

Нетрудно убедиться в том, что эти требования являются достаточными для однозначной интерпретации *операций обновления над представлениями*. Например, пусть имеется следующая спецификация запроса ([пример 17.9](#)):

```
SELECT EMP_SAL
FROM (SELECT EMP_SAL, DEPT_NO
 FROM EMP
 WHERE EMP_NAME = (SELECT EMP_NAME
 FROM EMP
 WHERE EMP_NO = 4425))
WHERE DEPT_NO <> 630;
```

Пример 17.9.

Эту спецификацию можно упростить до эквивалентной формулировки<sup>3</sup>:

```
SELECT EMP_SAL
FROM EMP
WHERE EMP_NAME = (SELECT EMP_NAME
 FROM EMP
 WHERE EMP_NO = 4425)
AND DEPT_NO <> 630;
```

Предположим, что с данной спецификацией запроса связано *представление* с именем **EMPSAL**. Тогда операция

```
UPDATE EMPSAL SET EMP_SAL = EMP_SAL - 1000.00;
```

*эквивалентна операции*

```
UPDATE EMP SET EMP_SAL = EMP_SAL - 1000.00
 WHERE EMP_NAME = (SELECT EMP_NAME
 FROM EMP
 WHERE EMP_NO = 4425)
 AND DEPT_NO <> 630;
```

Операция

```
DELETE FROM EMPSAL WHERE EMP_SAL > 20000.00;
```

*эквивалентна операции*

```
DELETE EMPSAL
 WHERE EMP_SAL > 20000.00 AND
 EMP_NAME = (SELECT EMP_NAME
 FROM EMP
 WHERE EMP_NO = 4425)
 AND DEPT_NO <> 630;
```

Операция вставки над *представлением* **EMPSAL**

```
INSERT INTO EMPSAL 25000.00;
```

трактруется как

```
INSERT INTO EMP
 ROW (DEFAULT, DEFAULT, DEFAULT, 25000.00, DEFAULT, DEFAULT);
```

Понятно, что такая операция будет отвергнута системой, потому что для столбца **EMP\_NO** таблицы **EMP** значения по умолчанию не определены (это *первичный ключ* таблицы, значения которого должны явно задаваться в любой операции вставки).

С другой стороны, условия *допустимости операций обновления, специфицированные в SQL/92*, не являются необходимыми. Например, над *представлением* **EMPMNG**, определенным над спецификацией запроса ("выбрать данные о служащих, являющихся руководителями отделов").

```
SELECT *
 FROM EMP
 WHERE EXISTS (SELECT *
 FROM DEPT
 WHERE DEPT_MNG = EMP_NO);
```

можно было бы совершенно корректно выполнять операции обновления (с некоторыми оговорками насчет операции вставки; см. ниже в этом разделе).

## **Представления, допускающие применение операций обновления, в стандарте SQL:1999**

В стандарте *SQL:1999* правила применимости операций обновления к спецификации запроса существенно уточнены.

### **Критерии применимости операций обновления**

Введены понятия *потенциальной применимости операций обновления*, *применимости операций обновления*, *простой применимости операций обновления* и *применимости операции вставки*. К спецификации запроса потенциально применимы операции обновления в том и только в том случае, когда выполняются следующие условия:

- в разделе **SELECT** спецификации запроса отсутствует ключевое слово **DISTINCT** ;
- элемент списка выборки раздела **SELECT**, состоящий из ссылки на некоторый столбец, не может присутствовать в этом списке более одного раза;
- в спецификации запроса отсутствуют разделы **GROUP BY** и **HAVING**.

Если выражение запросов отвечает условиям *потенциальной применимости операций обновления* и в его разделе **FROM** присутствует только одна ссылка на таблицу, то к каждому столбцу выражения запроса, соответствующему одному столбцу таблицы из раздела **FROM**, применимы операции обновления. Если выражение запроса отвечает условиям *потенциальной применимости операций обновления*, но в его разделе **FROM** присутствуют две или более ссылки на таблицы, то операции обновления применимы к столбцу выражения запросов только при выполнении следующих условий:

- столбец порождается из столбца только одной таблицы из раздела **FROM** ;
- эта таблица используется в выражении запросов таким образом, что сохраняются свойства ее первичного и всех возможных ключей.

Другими словами, к столбцу таблицы, которая отвечает условиям *потенциальной применимости операций обновления*, применимы операции обновления только в том случае, когда этот столбец может быть однозначно сопоставлен с единственным столбцом единственной таблицы, участвующей в выражении запроса, и каждая строка выражения запроса может быть однозначно сопоставлена с единственной строкой данной таблицы.

Выражение запросов удовлетворяет условию *применимости операций обновления*, если по крайней мере к одному столбцу выражения запросов применимы операции обновления. Выражение запросов удовлетворяет условию *простой применимости операций обновления*, если в разделе **FROM** выражения запросов содержится ссылка только на одну таблицу, и все столбцы выражения запросов удовлетворяют условию *применимости операций обновления*.

Выражение запросов удовлетворит условию *применимости операций вставки*, если оно удовлетворяет условию *применимости операций обновления*; каждая из таблиц, от которых зависит это выражение (т.е. таблиц, на которые имеются ссылки в разделе **FROM**), удовлетворяет условию *применимости операций вставки* и выражение запросов не содержит операций **UNION**, **INTERSECT** и **EXCEPT**. Конечно, это

определение базируется на том факте, что для любой базовой таблицы условие применимости операции вставки удовлетворяется (при наличии привилегии **INSERT**, см. следующую лекцию).

## Правила функциональных зависимостей

Приведенный набор правил является достаточно грубым. В стандарте *SQL:1999* он уточняется набором дополнительных правил, устанавливающих восприимчивость различных языковых конструкций к операциям обновления и вставки. В основе этих правил лежит понятие **функциональной зависимости** ( **Functional Dependency** - **FD**, см. лекцию 6). Полагая, что в целом понятие *функциональной зависимости* уже не должно вызывать у читателей каких-либо затруднений, приведем несколько дополнительных определений, требуемых для понимания подхода, используемого в *SQL:1999*.

- Пусть **S** обозначает некоторое множество столбцов таблицы **T**, а **SS** обозначает некоторое подмножество **S**. Тогда по первой аксиоме Армстронга (см. раздел "Функциональные зависимости" лекции 6) **SSS**. В терминологии *SQL:1999* эта **FD** называется **аксиоматической**. Все ФЗ, не являющиеся **аксиоматическими**, называются **неаксиоматическими**.
- Все **аксиоматические FD** являются **известными FD**. В стандарте определяются правила определения других **известных FD**. Кроме того, стандарт оставляет свободу для реализаций *SQL* в пополнении этой системы правил с целью нахождения **известных FD**, не специфицированных в стандарте.
- Если некоторый столбец **C1** виртуальной таблицы **T1** (порождаемой таблицы или представления) определяется путем ссылки на столбец **C2** некой другой (базовой или виртуальной) таблицы **T2**, на основе которой порождается **T1**, то **C1** является двойником **C2**. Более точно, **C1** является двойником **C2** в соответствии с таблицей **T2**.
- Понятие двойников расширяется на множества столбцов. Если некоторое множество столбцов **S1** виртуальной таблицы **T1** определяется (путем отображения "один-в-один") множеством столбцов **S2** определяющей таблицы **T2**, и каждый столбец из множества **S1** является двойником соответствующего столбца из множества **S2**, то **S1** называется двойником **S2** в соответствии с таблицей **T2**.
- Если ни в одном из столбцов **возможного ключа** (набора столбцов, специфицированного в неоткладываемом ограничении уникальности) не допускается наличие **неопределенных значений**, то это множество столбцов называется **BUC-множеством** (акроним **BUC** происходит от *Base table Unique Constraint*). Любое множество столбцов, являющееся двойником **BUC-множества**, также есть **BUC-множество**, так что это свойство распространяется через различные выражения, производящие виртуальные таблицы. Если имеются два множества столбцов **S1** и **S2**, такие, что **S1S2**, **S1S2**, и **S2** является **BUC-множеством**, то и **S1** является **BUC-множеством**. Могут существовать таблицы, у которых **BUC-множество** является пустым. Такая таблица может содержать не более одной строки<sup>4</sup>. С другой стороны, могут существовать таблицы, у которых вообще отсутствуют **BUC-множества**<sup>5</sup>.
- Множество столбцов, составляющих **первичный ключ** таблицы, называется ее **ВРК-множеством** (акроним **ВРК** происходит от *Base table Primary Key*). Понятно, что каждое **ВРК-множество** является **BUC-множеством**. Если имеются два множества столбцов **S1** и **S2**, такие, что **S1S2**, **S1S2**, и **S2** является **ВРК-множеством**, то и **S1** является **ВРК-множеством**. Подобно **BUC-множествам**, **ВРК-множества** могут быть пустыми.

На основе этих определений в стандарте *SQL:1999* устанавливаются правила *функциональных зависимостей* для 11 компонентов языка.

- **Базовые таблицы.** Если у таблицы имеется *первичный ключ*, то соответствующее множество столбцов образует *ВРК-множество* этой таблицы. Если у таблицы имеется не откладываемое ограничение уникальности и ни у одного столбца, указанного в этом ограничении, не допускается наличие *неопределенных значений*, то соответствующее множество столбцов является *ВУС-множеством*. Если множество столбцов **UCL** базовой таблицы - *ВУС-множество*, а **CT** обозначает все множество столбцов этой таблицы, то **FD UCLCT** представляет собой известную функциональную зависимость базовой таблицы.
- **Конструкторы табличных значений.** Поскольку для конструкторов табличных значений невозможно определять ограничения, в стандарте *SQL:1999* для них не специфицированы *ВУС-* и *ВРК-множества*. В стандарте не определяются *известные функциональные зависимости* для такого рода конструкций, отличные от *аксиоматических*. Однако стандарт допускает, чтобы реализации SQL включали дополнительные механизмы определения *известных функциональных зависимостей*.
- **Соединенные таблицы.** Если говорить о соединенных таблицах, получаемых в результате применения операций *естественного соединения* (**NATURAL JOIN**) или соединения с заданием списка имен столбцов, значения которых должны совпадать (**USING**), то понятно, что соединенная таблица будет содержать двойников из одной или двух исходных таблиц. Если обозначить через **S** некоторое множество столбцов результирующей таблицы, а через **CT** - все множество столбцов этой таблицы, то **S** является *ВРК-множеством* в том и только в том случае, когда имеет двойника в одной или обеих исходных таблицах. В таком случае во всех столбцах **S** не допускаются неопределенные значения, и **FD SCT** является известной функциональной зависимостью.

В стандарте определяется несколько правил, на основе которых устанавливаются *известные функциональные зависимости* соединенных таблиц, но здесь мы приведем только простейшее из этих правил. Если соединенная таблица производится на основе одной из двух указанных выше операций, то в первой таблице-источнике присутствует один или более столбцов, соответствующих одноименным столбцам второй таблицы-источника. Обозначим через **SLCC** список следующих выражений (элемент списка соответствует общему столбцу):

**COALESCE (t1.colname, t2.colname) AS colname**<sup>6</sup>

Пусть **JT** обозначает ключевые слова, определяющие тип соединения (**INNER**, **LEFT**, **RIGHT**, **FULL** и т.д.), и пусть **TN1** и **TN2** обозначают имена таблиц или (если они заданы) имена псевдонимов двух таблиц-источников соответственно. Обозначим через **IR** результат вычисления следующего выражения запросов:

**SELECT SLCC, T1\*, T2\* FROM T1 JT JOIN T2;**

Тогда, в соответствии с правилами SQL, дополнительными *известными функциональными зависимостями* являются следующие:

- если **JT** задает **INNER** или **LEFT**, то действует **FD COALESCE (T1.Ci, T2.Ci)T1.Ci** для всех *i* от единицы до числа столбцов в **IR** ;
- если **JT** задает **INNER** или **RIGHT**, то действует **FD COALESCE (T1.Ci, T2.Ci)T2.Ci** для всех *i* от единицы до числа столбцов в **IR**.

Обозначим через **SL** некоторый список выборки. Пусть:

- если все столбцы первой и второй таблиц-источников являются общими, то **SL** совпадает с **SLCC** ;
- если среди столбцов таблиц-источников нет общих столбцов, то **SL** состоит из списка столбцов первой таблицы-источника, за которым следует список столбцов второй таблицы-источника;
- если все столбцы первой таблицы-источника являются общими, но у второй таблицы-источника имеются необщие столбцы, то **SL** состоит из **SLCC**, за которым следует список необщих столбцов второй таблицы-источника;
- аналогично, если все столбцы второй таблицы-источника являются общими, но у первой таблицы-источника имеются не общие столбцы, то **SL** состоит из **SLCC**, за которым следует список не общих столбцов первой таблицы-источника;
- наконец, если среди столбцов первой таблицы-источника и среди столбцов второй таблицы-источника имеются необщие столбцы, то **SL** состоит из **SLCC**, за которым следует список необщих столбцов первой таблицы-источника, а далее располагается список не общих столбцов второй таблицы-источника.

Тогда, в соответствии со стандартом, известными функциональными зависимостями виртуальной таблицы, получаемой путем соединения, являются известные функциональные зависимости выражения

**SELECT SL FROM IR;**

- **Ссылки на таблицы.** Столбцы виртуальной таблицы, производимой по ссылке на таблицу, являются естественными двойниками столбцов таблицы, которая идентифицируется ссылкой. Поэтому *BUC*- и *ВРК*-множества результирующей таблицы являются двойниками *BUC*- и *ВРК*-множеств исходной таблицы, и известные функциональные зависимости результирующей таблицы получаются путем замены имен столбцов исходной таблицы на имена столбцов результирующей таблицы в известных функциональных зависимостях исходной таблицы.
- **Раздел FROM.** Описывая общую семантику оператора выборки, мы отмечали, что на первом шаге выполнения этого оператора производится (виртуальная) таблица, являющаяся расширенным декартовым произведением всех таблиц, специфицированных в разделе **FROM**. Поэтому в стандарте SQL естественным образом формулируются следующие правила. Если в списке ссылок на таблицы раздела **FROM** содержится всего одна ссылка, то *BUC*- и *ВРК*-множества результирующей таблицы являются двойниками *BUC*- и *ВРК*-множеств исходной таблицы. Если в списке раздела **FROM** содержатся две или более ссылки на таблицы, то, в соответствии со стандартом, *BUC*- и *ВРК*-множества результирующей таблицы не определены. Известные функциональные зависимости результирующей таблицы состоят из известных функциональных зависимостей каждой таблицы, специфицированной в разделе **FROM**.
- **Раздел WHERE.** В стандарте содержится набор правил, позволяющих определить *BUC*- и *ВРК*-множества результирующей таблицы этого раздела<sup>7</sup>, а также известные функциональные зависимости результирующей таблицы. Правила основываются на особенностях поведения предиката сравнения по равенству и логической операции **AND**.
- **Раздел GROUP BY.** Для определения *BUC*- и *ВРК*-множеств и известных функциональных зависимостей результирующей таблицы раздела **GROUP BY** требуется фактическое образование в результирующей таблице нового столбца, значения которого

могли бы каким-то образом идентифицировать строки исходной таблицы, образующие группы сгруппированной таблицы.

- **Раздел *HAVING*.** *BUC-* и *BPK-*множества и известные функциональные зависимости результирующей таблицы раздела *HAVING* получаются из соответствующих множеств и FD таблицы, к которой применяется этот раздел<sup>8</sup>, на основе правил, связанных с условным выражением раздела *HAVING* (как и в случае условия раздела *WHERE*, в данных правилах учитываются операции сравнения по равенству и логические операции *AND* ).
- **Раздел *SELECT*.** На определение *BUC-* и *BPK-*множеств и известных функциональных зависимостей результата спецификации запроса влияет наличие в списке выборки выражений ( *value\_expression* ), отличных от ссылок на столбцы.
- **Выражение запроса.** На определение *BUC-* и *BPK-*множеств и известных функциональных зависимостей результата выражения запроса влияет наличие в этом выражении операций *UNION*, *INTERSECT* и *EXCEPT*. В стандарте отсутствуют какие-либо правила для определения функциональных зависимостей в результатах рекурсивных запросов. Отмечается лишь возможность введения таких правил в реализациях.

#### Раздел *WITH CHECK OPTION* определения представления

Пусть в базе данных имеется упрощенная таблица *EMP*, содержащая следующее множество строк (как в примере с *GROUP BY ROLLUP* разделе "Возможности формулирования аналитических запросов" лекции 16).

Предположим, что в базе данных имеется представление *RICH\_EMP*, определенное следующим образом:

| <i>EMP</i>    |                |                  |                |
|---------------|----------------|------------------|----------------|
| <i>EMP_NO</i> | <i>DEPT_NO</i> | <i>EMP_BDATE</i> | <i>EMP_SAL</i> |
| 2440          | 1              | 1950             | 15000.00       |
| 2441          | 1              | 1950             | 16000.00       |
| 2442          | 1              | 1960             | 14000.00       |
| 2443          | 1              | 1960             | 19000.00       |
| 2444          | 2              | 1950             | 17000.00       |
| 2445          | 2              | 1950             | 16000.00       |

|      |   |      |          |
|------|---|------|----------|
| 2446 | 2 | 1960 | 14000.00 |
| 2447 | 2 | 1960 | 20000.00 |
| 2448 | 3 | 1950 | 18000.00 |
| 2449 | 3 | 1950 | 13000.00 |
| 2450 | 3 | 1960 | 21000.00 |
| 2451 | 3 | 1960 | 22000.00 |

```
CREATE VIEW RICH_EMP AS
SELECT *
FROM EMP
WHERE EMP_SAL > 18000.00;
```

Понятно, что в соответствии с правилами SQL (и здравым смыслом) над этим *представлением* можно выполнять операции обновления. Как видно, в таблице *EMP* содержится строка, которая соответствует служащему с номером **2447**, получающему зарплату в размере **20000** руб. Естественно, эта строка будет присутствовать в *виртуальной таблице* *RICH\_EMP*. Поэтому можно было бы выполнить, например, операцию

```
UPDATE RICH_EMP
SET EMP_SAL = EMP_SAL - 3000
WHERE EMP_NO = 4452;
```

Но если выполнение такой операции действительно допускается, то в результате строка, соответствующая служащему с номером **2447**, исчезнет из *виртуальной таблицы* *RICH\_EMP*! Аналогичный эффект возникнет при выполнении операции вставки

```
INSERT INTO RICH_EMP (EMP_NO) 2452;
```

В базовой *таблице* *EMP* появится строка, в которой значением столбца *EMP\_NO* будет **2452**, а значения остальных столбцов будут установлены по умолчанию. В частности, значением столбца *EMP\_SAL* будет **10000.00**. Тем самым, если подобная операция вставки действительно допустима, то мы вставили в *виртуальную таблицу* *RICH\_EMP* строку, которую в этой *виртуальной таблице* увидеть невозможно.

Чтобы избежать такого противоречивого поведения представляемых таблиц, нужно включать в определение *представления* раздел *WITH CHECK OPTION*. При наличии этого раздела до реального выполнения операций модификации или вставки строк через *представление* для каждой строки будет проверяться, что она соответствует условиям *представления*. Если данное условие не выполняется хотя бы для

одной *модифицируемой* или *вставляемой* строки, то операция полностью отвергается. В некотором смысле (при наличии раздела **WITH CHECK OPTION**) условие выборки, содержащееся в выражении запросов *представления*, можно считать *ограничением целостности* этого *представления*.

## Режимы проверки CASCADED и LOCAL

Вспомним теперь, что в полном виде синтаксис раздела **WITH CHECK OPTION** может включать ключевые слова **CASCADED** или **LOCAL**. Обсудим их смысл. Предположим, что *представление V2* определяется над *представлением V1* следующим образом:

```
CREATE VIEW V2 AS
 SELECT ...
 FROM V1
 WHERE ...
 [WITH [CASCADED | LOCAL] CHECK OPTION]
```

Пусть над **V2** выполняется некоторая операция **O** обновления базы данных. Тогда:

- если *представление V2* определялось без раздела **WITH CHECK OPTION**, то при выполнении операции **O** будут проверяться все условия, определяющие *ограничения целостности V1* (если в определении **V1** присутствовал раздел **WITH CHECK OPTION**), но никаким образом не будут учитываться условия выборки, содержащееся в выражении запросов *представления V2*;
- если в определении *представления V2* содержался раздел **WITH LOCAL CHECK OPTION**, то при выполнении операции **O** будут проверяться все условия, определяющие *ограничения целостности V1*, и все условия, содержащееся в выражении запросов *представления V2*;
- наконец, если в определении *представления V2* содержался раздел **WITH CASCADED CHECK OPTION**, то при выполнении операции **O** будут проверяться все условия, определяющие *ограничения целостности V1* (так, как если бы в определении **V1** присутствовал раздел **WITH CASCADED CHECK OPTION**). Тем самым, будут проверяться все *ограничения целостности*, установленные для всех *базовых таблиц*, на которых основывается определение **V1**; все условия всех *представлений*, определенных над этими *базовыми таблицами*; и, конечно, все условия, содержащиеся в выражении запросов *представления V2*.

## Примеры результатов действия раздела WITH CHECK OPTION

Чтобы пояснить результаты действия раздела **WITH CHECK OPTION**, допустим, что в базе данных присутствуют определения двух *представлений* **MIDDLE\_RICH\_EMP** и **MORE\_RICH\_EMP**:

```
CREATE VIEW MIDDLE_RICH_EMP AS
 SELECT *
 FROM EMP
 WHERE EMP_SAL < 20000.00
 [WITH [CASCADED | LOCAL] CHECK OPTION];
CREATE VIEW MORE_RICH_EMP AS
 SELECT *
 FROM MIDDLE_RICH_EMP
```

WHERE EMP\_SAL > 18000.00  
 [ WITH [ CASCADED | LOCAL ] CHECK OPTION ];

Очевидно, что в тело (материализованного) представления **MIDDLE\_RICH\_EMP** будут входить следующие строки базовой таблицы **EMP**:

| MIDDLE_RICH_EMP |         |           |          |
|-----------------|---------|-----------|----------|
| EMP_NO          | DEPT_NO | EMP_BDATE | EMP_SAL  |
| 2440            | 1       | 1950      | 15000.00 |
| 2441            | 1       | 1950      | 16000.00 |
| 2442            | 1       | 1960      | 14000.00 |
| 2443            | 1       | 1960      | 19000.00 |
| 2444            | 2       | 1950      | 17000.00 |
| 2445            | 2       | 1950      | 16000.00 |
| 2446            | 2       | 1960      | 14000.00 |
| 2448            | 3       | 1950      | 18000.00 |
| 2449            | 3       | 1950      | 13000.00 |

В тело (материализованного) представления **MORE\_RICH\_EMP** будут входить следующие строки представляемой таблицы **MIDDLE\_RICH\_EMP**:

| MORE_RICH_EMP |         |           |         |
|---------------|---------|-----------|---------|
| EMP_NO        | DEPT_NO | EMP_BDATE | EMP_SAL |

|      |   |      |          |
|------|---|------|----------|
| 2443 | 1 | 1960 | 19000.00 |
|------|---|------|----------|

В каждом из представлений **MIDDLE\_RICH\_EMP** и **MORE\_RICH\_EMP** может отсутствовать или присутствовать (в одном из двух видов) раздел **WITH CHECK OPTION**. В совокупности возможен один из девяти случаев:

| <b>MORE_RICH_EMP</b>   |             |              |                 |
|------------------------|-------------|--------------|-----------------|
|                        | <b>none</b> | <b>LOCAL</b> | <b>CASCADED</b> |
| <b>MIDDLE_RICH_EMP</b> |             |              |                 |
| <b>none</b>            | Случай 1    | Случай 2     | Случай 3        |
| <b>LOCAL</b>           | Случай 4    | Случай 5     | Случай 6        |
| <b>CASCADED</b>        | Случай 7    | Случай 8     | Случай 9        |

Чтобы рассмотреть каждый из возможных случаев по отдельности, обсудим, что будет происходить в каждом случае при выполнении следующих двух операций модификации строк (будем называть эти операции **U1** и **U2** соответственно)<sup>9</sup>:

```
UPDATE MORE_RICH_EMP
SET EMP_SAL = EMP_SAL + 7000.00;
UPDATE MORE_RICH_EMP
SET EMP_SAL = EMP_SAL - 7000.00;
```

**Случай 1.** Ни в одном из представлений не содержится раздел **WITH CHECK OPTION**.

Первый неожиданный результат состоит в том, что после выполнения операции **U1** тело представления **MORE\_RICH\_EMP** оказывается пустым. Действительно, у единственной строки таблицы **EMP** (со значением **EMP\_NO**, равным 2443), одновременно удовлетворяющей условиям обоих представлений, столбец **EMP\_SAL** принимает значение 26000.00. После этого строка перестает удовлетворять условию представления **MIDDLE\_RICH\_EMP** и исчезает из результирующей таблицы **MORE\_RICH\_EMP**. Этот результат может быть особенно неожиданным для пользователей базы данных, которым известно, что условие представления **MORE\_RICH\_EMP** имеет вид **EMP\_SAL > 18000.00**, и соблюдение этого условия должно сохраняться при увеличении размера зарплаты.

Выполнение операции **U2** также приведет к опустошению тела **MORE\_RICH\_EMP** (в базовой таблице **EMP** не останется ни одной строки, удовлетворяющей условию этого представления). Возможно, это будет достаточно естественно для пользователей представления **MORE\_RICH\_EMP**, которым известно

условие *представления*, но те, кто работает с *представлением* **MIDDLE\_RICH\_EMP**, с удивлением обнаружат в теле результирующей таблицы новые строки.

**Случай 2.** В определении *представления* **MIDDLE\_RICH\_EMP** содержится раздел **WITH LOCAL CHECK OPTION**, а в определении **MORE\_RICH\_EMP** раздел **WITH CHECK OPTION** отсутствует.

В этом случае, в соответствии с первыми двумя правилами проверки корректности выполнения операций обновления над *представлениями*, операция **U1** должна быть отвергнута системой (поскольку ее выполнение нарушает условие *представления* **MIDDLE\_RICH\_EMP**). Но заметим, что такое поведение системы будет совершенно неожиданным и непонятным для тех *пользователей базы данных*, которым известно только определение "верхнего" *представления* **MORE\_RICH\_EMP**, поскольку операция **U1** явно не может нарушить видимое ими ограничение.

С другой стороны, операция **U2** будет успешно выполнена и по-прежнему приведет к опустошению тела результирующей таблицы *представления* **MORE\_RICH\_EMP**.

**Случай 3.** В определении *представления* **MIDDLE\_RICH\_EMP** содержится раздел **WITH CASCADED CHECK OPTION**, а в определении **MORE\_RICH\_EMP** раздел **WITH CHECK OPTION** отсутствует.

В этой ситуации будут проверяться условия, содержащиеся в определении *представления* **MIDDLE\_RICH\_EMP**, а также все *ограничения целостности* таблицы **EMP** и всех других *представлений*, определенных над этой базовой таблицей. В результате операция **U1** будет отвергнута системой, а операция **U2** будет "успешно" выполнена. Другими словами, повторится Случай 2.

**Случай 4.** В определении *представления* **MIDDLE\_RICH\_EMP** раздел **WITH CHECK OPTION** отсутствует, а в определении **MORE\_RICH\_EMP** содержится раздел **WITH LOCAL CHECK OPTION**.

Понятно, что в этом варианте операция **U2** не сработает (ее выполнение не будет допущено условием "*ограничения целостности*" *представления* **MORE\_RICH\_EMP**). Но операция **U1** (увеличение размера зарплаты служащих) будет успешно выполнена, поскольку она не противоречит локальным ограничениям *представления* **MORE\_RICH\_EMP**.

**Случай 5.** В определениях *представлений* **MIDDLE\_RICH\_EMP** и **MORE\_RICH\_EMP** содержится раздел **WITH LOCAL CHECK OPTION**.

Выполнение обеих операций **U1** и **U2** будет справедливо отвергнуто. На первый взгляд все в порядке. Но если над *представлением* **MORE\_RICH\_EMP** будет определено еще одно *представление* **V**, то мы можем получить ситуацию Случая 2, где **V** будет играть роль **MORE\_RICH\_EMP**, а **MIDDLE\_RICH\_EMP** - роль **MORE\_RICH\_EMP**.

**Случай 6.** В определении *представления* **MIDDLE\_RICH\_EMP** содержится раздел **WITH CASCADED CHECK OPTION**, а в определении **MORE\_RICH\_EMP** содержится раздел **WITH LOCAL CHECK OPTION**.

Снова, если над представлением **MORE\_RICH\_EMP** будет определено еще одно представление **V**, то мы можем попасть в ситуацию Случая 2, где **V** будет играть роль **MORE\_RICH\_EMP**, а **MIDDLE\_RICH\_EMP** - роль **MORE\_RICH\_EMP**.

**Случай 7.** В определении представления **MIDDLE\_RICH\_EMP** раздел **WITH CHECK OPTION** отсутствует, а в определении **MORE\_RICH\_EMP** содержится раздел **WITH CASCADED CHECK OPTION**.

Если над представлением **MORE\_RICH\_EMP** будет определено еще одно представление **V**, то мы можем попасть в ситуацию Случая 3, где **V** будет играть роль **MORE\_RICH\_EMP**, а **MIDDLE\_RICH\_EMP** - роль **MORE\_RICH\_EMP**.

**Случай 8.** В определении представления **MIDDLE\_RICH\_EMP** содержится раздел **WITH LOCAL CHECK OPTION**, а в определении **MORE\_RICH\_EMP** - раздел **WITH CASCADED CHECK OPTION**.

Если над представлением **MORE\_RICH\_EMP** будет определено еще одно представление **V**, то мы можем получить ситуацию Случая 3, где **V** будет играть роль **MORE\_RICH\_EMP**, а **MIDDLE\_RICH\_EMP** - роль **MORE\_RICH\_EMP**.

**Случай 9.** В определениях представлений **MIDDLE\_RICH\_EMP** и **MORE\_RICH\_EMP** содержится раздел **WITH CASCADED CHECK OPTION**.

Только в этом случае операции обновления будут выполняться корректно, независимо от того, имеются ли в базе данных представления, определенные над **MORE\_RICH\_EMP** или между **MORE\_RICH\_EMP**, **MIDDLE\_RICH\_EMP** и **EMP**.

Очевидный вывод из приведенного анализа заключается в том, что единственным способом обеспечить корректность выполнения операций обновления через представления (допускающие операции обновления) является включение в определение каждого представления раздела **WITH CASCADED CHECK OPTION**. В этом случае поведение системы будет оставаться корректным при введении дополнительных представлений над представлением **MORE\_RICH\_EMP**, между представлениями **MORE\_RICH\_EMP** и **MIDDLE\_RICH\_EMP** или между представлением **MIDDLE\_RICH\_EMP** и базовой таблицей **EMP**, если в определениях всех этих представлений присутствует раздел **WITH CASCADED CHECK OPTION**.

## Исторический очерк

Завершим обсуждение возможностей применения операций обновления к виртуальным таблицам небольшим экскурсом в историю. На протяжении более чем тридцатилетней истории реляционных баз данных вопрос о возможности однозначной интерпретации операций обновления баз данных через виртуальные таблицы интересовал многих исследователей. Причины этого интереса состоят в следующем.

Во-первых, как отмечалось в лекции 3, одной из наиболее привлекательных черт реляционной алгебры является замкнутость относительно понятия отношения. В любой алгебраической операции, операндом которой является отношение, в качестве операнда можно использовать алгебраическое выражение. С другой стороны, имеется явное неравноправие по отношению к операциям обновления. Мы можем вставлять,

модифицировать и удалять *кортежи* в базовых отношениях, но не можем (в общем случае) применять эти операции к алгебраическим выражениям. Хотелось максимальным образом устранить подобное неравноправие.

Во-вторых, на первый взгляд задача не является слишком трудной (по крайней мере, если оставаться в пределах *реляционной алгебры*). Действительно, базовых операций совсем немного, и каждая базовая операция очень проста.

К сожалению, это ощущение простоты проблемы оказалось обманчивым. Было выполнено множество исследований, опубликовано множество статей (нам кажется нецелесообразным приводить список этих статей в данном курсе), но так и не удалось обнаружить полное множество алгебраических выражений, для которых возможна однозначная интерпретация операций обновления. На мой взгляд, данная ситуация оказала заметное влияние на подход к решению проблемы *применимости операций обновления к виртуальным таблицам*, которым руководствуются разработчики языка SQL.

В двух первых *международных стандартах* (SQL/89 и SQL/92) к виду таких виртуальных таблиц предъявлялись чрезмерно строгие требования. Это показывают даже те простые примеры, которые приводились в начале данного раздела. И конечно, наличие таких ограничений в *стандарте языка* приводило к тому, что в реализациях SQL появлялось много расширений, которые поддерживались только отдельными компаниями-производителями СУБД. Создается впечатление, что когда более десяти лет назад был инициирован проект нового стандарта SQL-3 (который в конце концов привел к появлению SQL:1999), разработчики находились в состоянии растерянности.

Кстати, одна из идей, включавшихся в ранние варианты проекта SQL-3, состояла в том, чтобы расширить определение представляемой таблицы средствами, позволяющими явно специфицировать действия, которые нужно предпринимать при выполнении над *представлением* операций *INSERT*, *UPDATE* и *DELETE*. Другими словами, предлагалось переложить решение проблемы на плечи пользователей СУБД. Конечно, это радикальный подход, но, с другой стороны, он мог бы привести к полной анархии.

Как можно заметить, в официально принятом *стандарте SQL:1999* используется некоторый компромиссный подход. В стандарте не фиксируются жесткие правила, ограничивающие вид виртуальных таблиц, к которым *применимы операции обновления*. Вместо этого сформулирован ряд рекомендаций, которыми следует руководствоваться производителям СУБД. Нельзя утверждать, что такое решение является идеальным, но более удачного решения найти не удалось.

## Операции обновления баз данных и механизм триггеров

Термин *триггер* в контексте *реляционных баз данных* был введен в обиход участниками проекта System R (лекции 11). В терминологии этого проекта *триггером* называлась хранимая в базе данных процедура, автоматически вызываемая СУБД при возникновении соответствующих условий. При определении *триггера* указывались два условия его применимости - общее условие (имя отношения и тип *операции* манипулирования данными) и конкретное условие (*логическое выражение*, построенное по правилам, близким к правилам *ограничений целостности*), а также действие, которое должно быть выполнено над *БД* при наличии условий применимости.

Конечно, термин *триггер* в данном контексте является жаргонным. Но, с другой стороны, он достаточно точно соответствует ситуации: для применения процедуры должны быть произведены "возбуждающие" ее действия. Как отмечалось в лекции 11, после завершения проекта System R на протяжении более десяти лет *триггеры* не поддерживались ни в одной коммерческой SQL-ориентированной СУБД. Но затем практически во всех ведущих СУБД механизм *триггеров* в том или ином виде был реализован.

В стандарте же языка SQL спецификации *триггеров* отсутствовали до принятия стандарта SQL:1999. По словам главного редактора стандартов SQL/92 и SQL:1999 Джима Мелтона, эта спецификация была уже полностью готова к моменту принятия SQL/92 и не вошла в текст стандарта только по причине ограниченности его объема. Однако, как мне кажется, этому препятствовали и расхождения в подходах, существовавшие между основными компаниями-производителями СУБД.

Заметим, что альтернативным термином по отношению к базам данных, содержащим *триггерные процедуры*, является термин **активная база данных**. Наверное, этот термин более точен, поскольку действительно речь идет о базах данных, содержащих процедуры, которые автоматически вызываются при срабатывании связанных с ними правил. Однако в обиходе пользователей SQL-ориентированных СУБД по-прежнему более распространен термин *триггер*.

### Понятие триггера в SQL:1999

В языке обеспечиваются возможности определения *триггеров*, которые вызываются ("срабатывают") при вставке одной или нескольких строк в указанную таблицу, при модификации одной или нескольких строк в указанной таблице или при удалении одной или нескольких строк из указанной таблицы. Вообще говоря, *триггер* может производить любое действие, необходимое для соответствующего приложения. Можно определить *триггеры*, срабатывающие по одному разу для операций *INSERT*, *UPDATE* или *DELETE*, но существует и возможность определения *триггеров*, вызываемых при вставке, модификации или удалении каждой отдельной строки. Таблица, с которой связывается определение *триггера*, называется **предметной таблицей** ( *subject table* ), а оператор SQL, выполнение которого приводит к срабатыванию *триггера*, мы будем называть **инициирующим** ( *triggering SQL statement* ).

*Триггеры* могут срабатывать **после** и **до** реального выполнения *инициирующего оператора SQL*. В теле *триггера* допускается доступ к значениям вставляемых, *модифицируемых* и удаляемых строк. В случае операции модификации возможен доступ к значениям строк до модификации и к значениям после модификации. В соответствии со стандартом SQL:1999 любой *триггер* ассоциируется только с одной базовой таблицей. Не допускается определение *триггеров* над представлениями<sup>10</sup>.

Можно придумать различные способы полезного применения механизма *триггеров*, но принято считать, что основными областями использования этого механизма являются следующие.

- **Журнализация и аудит.** С помощью *триггеров* можно отслеживать изменения таблиц, для которых требуется поддержка повышенного уровня безопасности. Данные об изменении таблиц могут сохраняться в других таблицах и включать,

например, *идентификатор пользователя*, от имени которого выполнялась операция обновления; временную метку операции обновления; сами обновляемые данные и т. д.

- **Согласование и очистка данных.** С любым простым оператором SQL, обновляющим некоторую таблицу, можно связать *триггеры*, производящие соответствующие обновления других таблиц. Например, с операцией вставки новой строки в таблицу *EMP* (прием на работу нового служащего) можно было связать *триггер*, модифицирующий значения столбцов *DEPT\_EMP\_NO* и *DEPT\_TOTAL\_SAL* <sup>11</sup> строки таблицы *DEPT* со значением столбца *DEPT\_NO*, которое соответствует номеру отдела нового служащего.
- **Операции, не связанные с изменением базы данных.** В *триггерах* могут выполняться не только операции обновления базы данных. Стандарт SQL позволяет определять хранимые процедуры (которые могут вызываться из *триггеров*), посылающие электронную почту, печатающие документы и т. д.

### Синтаксис определения триггеров и типы триггеров

Для более подробного обсуждения механизма *триггеров* в *SQL:1999* необходимо ввести набор синтаксических правил:

```
trigger_definition ::=
 CREATE TRIGGER trigger_name
 { BEFORE | AFTER }
 { INSERT | DELETE | UPDATE [OF column_commalist] }
 ON table_name [REFERENCING
old_or_new_values_alias_list]
 triggered_action
triggered_action ::=
 [FOR EACH { ROW | STATEMENT }]
 [WHEN left_paren conditional_expression right_paren]
 triggered_SQL_statement
triggered_SQL_statement ::= SQL_procedure_statement
 | BEGIN ATOMIC
 SQL_procedure_statement_semicolonlist
 END
old_or_new_values_alias ::= OLD [ROW] [AS] correlation_name
 | NEW [ROW] [AS] correlation_name
 | OLD TABLE [AS] identifier
 | NEW TABLE [AS] identifier
```

Естественно, в языке имеется и конструкция, отменяющая определение *триггера*:

```
DROP TRIGGER trigger_name.
```

(Конструкция *ALTER TRIGGER* в языке SQL не поддерживается.)

Как мы видим, синтаксические правила допускают несколько разновидностей определения *триггера*. Кратко обсудим эти разновидности.

### Триггеры BEFORE и AFTER

Если в определении *триггера* указано ключевое слово **BEFORE**, то *триггер* будет срабатывать непосредственно до выполнения операции обновления *базовой таблицы* соответствующим *инициирующим оператором SQL*. При задании ключевого слова **AFTER** *триггер* будет вызываться немедленно после выполнения *инициирующего оператора*.

## Триггеры INSERT, UPDATE и DELETE

Выбор одного из этих ключевых слов при определении *триггера* указывает на природу события, которое должно приводить к срабатыванию *триггера*. При задании ключевого слова **INSERT** к срабатыванию *триггера* может привести только выполнение операции вставки строк в *предметную таблицу*. Если указываются ключевые слова **UPDATE** или **DELETE**, то число возможных событий, приводящих к срабатыванию *триггера*, возрастает. Кроме явных операций модификации строк *предметной таблицы* или удаления из нее строк к срабатыванию *триггера* могут привести *ссылочные действия* (см. раздел "Средства определения, изменения и ликвидации базовых таблиц" лекции 12).

Заметим, что в *стандарте SQL:1999* отсутствует возможность определения *триггеров*, для которых событием было бы выполнение операции выборки из *предметной таблицы*. Разработчики стандарта сочли, что область применения *триггеров* такого рода чересчур узка (трудно придумать какое-либо применение, кроме как для *журнализации* и аудита).

## Триггеры ROW и STATEMENT

Если в определении *триггера* присутствует конструкция **FOR EACH ROW**, то *триггер* будет вызываться для каждой строки *предметной таблицы*, обновляемой *инициирующим SQL-оператором*. Если же задано **FOR EACH STATEMENT** (или явная спецификация **FOR EACH** отсутствует), то *триггер* сработает один раз на всем протяжении процесса выполнения *инициирующего SQL-оператора*.

## Раздел WHEN

Включение в определение *триггера* раздела **WHEN** с соответствующим *условным выражением* позволяет более точно специфицировать условие применимости *триггера*. Вычисление *условного выражения* производится над строками *предметной таблицы*, и *триггер* срабатывает только в том случае, когда значением *условного выражения* является **true**. Понятно, что виды и интерпретация *логических выражений*, допускаемых в разделе **WHEN**, различаются у *триггеров* с **FOR EACH ROW** и у *триггеров* с **FOR EACH STATEMENT**. В первом случае *условное выражение* вычисляется для одной строки, которая должна быть обновлена *инициирующим SQL-оператором*. Во втором - *условное выражение* вычисляется для всей *предметной таблицы* целиком и, по всей видимости, должно базироваться на "кванторных" *предикатах*. Следует также понимать, что вычисление условия раздела **WHEN** данного *триггера* производится только в том случае, если произошло событие срабатывания *триггера*.

## Тело триггера

Операции, которые должны быть выполнены при срабатывании *триггера*, специфицируются в синтаксической конструкции **triggered\_SQL\_statement** (будем

называть ее *инициируемым SQL-оператором*). Как видно из синтаксических правил, возможны два вида построения этой конструкции: в виде одиночного оператора SQL и в виде списка операторов со скобками **BEGIN ATOMIC** и **END**.

Недоумение читателей может вызвать неуточненная конструкция **SQL\_procedure\_statement**. Постараемся объяснить ее происхождение и смысл. Дело в том, что в *стандарте SQL:1999* определено процедурное расширение SQL, называемое *SQL/PSM* (от *Persistent Stored Modules*). Это достаточно большой язык, который мы не будем подробно рассматривать в этом курсе лекций<sup>12</sup>. Тем не менее для понимания синтаксиса определения *триггеров* необходимо отметить, что: (a) *SQL/PSM* включает основные операторы SQL, связанные с обновлением данных; (b) язык является вычислительно полным, т.е. включает развитые средства вычислений; (c) в языке содержатся средства определения и вызова функций и процедур<sup>13</sup> и (d) *SQL/PSM* содержит стандартный комплект *управляющих конструкций* - циклы, *ветвления* разных типов и т. д. Тем самым, **SQL\_procedure\_statement** - это любая процедура, определенная на языке *SQL/PSM*<sup>14</sup>. В частности, эта процедура может представлять собой *оператор SQL* обновления базы данных.

Обсудим теперь, откуда возникает потребность в *составном иницируемом SQL-операторе*. Дело в том, что на практике при определении *триггеров* в качестве **SQL\_procedure\_statement** чаще всего используются операторы SQL обновления базы данных. Иногда (и мы покажем это на примере) для корректного определения функциональности *триггера* одного оператора не хватает, а в SQL отсутствует возможность определения *составных операторов*. Поэтому допускается использование средств определения *составных операторов*, присутствующих в *SQL/PSM* (**BEGIN ATOMIC** и **END**).

Для иллюстрации случая, когда при определении *триггера* достаточно специфицировать один *оператор SQL*, приведем пример определения *триггера*, условием срабатывания которого является выполнение операции вставки новой строки в таблицу **EMP** (прием на работу нового служащего). Если значение столбца **DEPT\_NO** в очередной вставляемой строке отлично от **NULL**, то *триггер* должным образом модифицирует значения столбцов **DEPT\_EMP\_NO** и **DEPT\_TOTAL\_SAL** строки таблицы **DEPT** со значением столбца **DEPT\_NO**, которое соответствует номеру отдела нового служащего ([пример 17.10](#)):

```
CREATE TRIGGER DEPT_CORRECTION AFTER INSERT ON EMP
FOR EACH ROW
WHEN (EMP.DEPT_NO IS NOT NULL)
UPDATE DEPT SET
DEPT_EMP_NO = DEPT_EMP_NO + 1,
DEPT_TOTAL_SAL = DEPT_TOTAL_SAL + EMP_SAL
WHERE DEPT.DEPT_NO = EMP.DEPT_NO;
```

Пример 17.10.

Теперь предположим, что при увольнении служащего (удалении строки из таблицы **EMP**) мы хотим не только должным образом модифицировать таблицу **DEPT**, но и сохранять (с целью аудита) данные об уволенном служащем в таблице **EMP\_DISMISSED**<sup>15</sup>:

EMP\_DISMISSED:

EMP\_NO : EMP\_NO

EMP\_NAME : VARCHAR

DEPT\_NO : DEPT\_NO

Определение соответствующего *триггера* могло бы выглядеть следующим образом (пример 17.11):

```
CREATE TRIGGER EMP_DISMISSION AFTER DELETE ON EMP
FOR EACH ROW
BEGIN ATOMIC
 INSERT INTO EMP_DISMISSED
 ROW (EMP.EMP_NO, EMP.EMP_NAME, EMP.DEPT_NO);
 UPDATE DEPT SET
 DEPT_EMP_NO = DEPT_EMP_NO - 1,
 DEPT_TOTAL_SAL = DEPT_TOTAL_SAL - EMP_SAL
 WHERE DEPT.DEPT_NO = EMP.DEPT_NO
END;
```

Пример 17.11.

### Выполнение триггеров

При выполнении каждого *триггера* система устанавливает *контекст выполнения триггера*. Выполнение любого оператора SQL, обновляющего базовую таблицу базы данных, может привести к срабатыванию одного или нескольких *триггеров*, а выполнение операторов SQL, содержащихся в *триггерах*, может привести к обновлению других базовых таблиц. Эти "внутритриггерные" (инициируемые) операторы выполняются в *контексте текущего триггера*, но их выполнение может привести к срабатыванию других *триггеров*. Для каждого из "вторичных" *триггеров* образуется собственный *контекст выполнения*, позволяющий определить их действия точно и независимо от действий первого набора *триггеров*. Выполнение *вторичных триггеров* может привести к срабатыванию "третичных" *триггеров* и т.д. - допускается произвольная глубина вложенности. Для каждого *триггера* на каждом уровне образуется собственный *контекст*.

*Контекст выполнения триггера* всегда является *атомарным*, т.е. *инициируемый SQL-оператор* либо успешно завершается, либо результаты его действия гарантированно отсутствуют в базе данных.

Обсудим понятие *контекста триггера* немного более подробно. Предположим, что в нашей базе данных *EMP-DEPT-PRO* должно поддерживаться правило, в соответствии с которым каждый служащий, становящийся *руководителем проекта*, автоматически получает прибавку к *заработной плате* в 10 000 руб. (Для простоты будем считать, что снятие служащего с должности *руководителя проекта* не приводит к автоматическому

изменению его зарплаты и что для каждого служащего, являющегося *руководителем проекта*, определен номер отдела, в котором он работает.) Тогда мы могли бы определить *триггер* **CHANGE\_MNG\_NO** следующим образом:

```
CREATE TRIGGER CHANGE_MNG_NO AFTER UPDATE OF PRO_MNG ON PRO
FOR EACH ROW
UPDATE EMP SET EMP_SAL = EMP_SAL + 10000.00
WHERE EMP_NO = PRO_MNG;
```

Но очевидно, что для поддержания корректности данных в таблице **DEPT** нам требуется *триггер*, условием срабатывания которого было бы изменение значений столбца **EMP\_SAL** в таблице **EMP**. Определим соответствующий *триггер* **DEPT\_CORRECTION\_1**:

```
CREATE TRIGGER DEPT_CORRECTION_1
AFTER UPDATE OF EMP_SAL ON EMP
REFERENCING OLD ROW AS OLD_EMP NEW ROW AS NEW_EMP
FOR EACH ROW
UPDATE DEPT SET
DEPT_TOTAL_SAL =
DEPT_TOTAL_SAL + NEW_EMP.EMP_SAL -
OLD_EMP.EMP_SAL
WHERE EMP.DEPT_NO = DEPT.DEPT_NO;
```

Пусть теперь выполняется операция

```
UPDATE PRO SET PRO_MNG = 4455
WHERE PRO_NO = 554;
```

Сразу после выполнения этой операции сработает *триггер* **CHANGE\_MNG\_NO**. Этот *триггер* будет выполняться в контексте, который мы для удобства назовем контекстом **CMN**. Заметим, что исходный оператор модификации в действительности изменяет только одну строку таблицы **PRO**, но *триггеру* **CHANGE\_MNG\_NO** это неизвестно, и он будет работать так, как если бы изменялось произвольное число строк таблицы **PRO**.

Выполнение операции модификации таблицы **EMP** приведет к срабатыванию *триггера* **DEPT\_CORRECTION\_1**. В этот момент контекст **CMN** будет "упрятан в стек", образуется и станет активным контекст следующего *триггера* - контекст **DR1**. После завершения выполнения этого *триггера* контекст **DR1** больше не требуется, и он ликвидируется, а из стека восстанавливается контекст **CMN**, в котором и будет завершено выполнение *триггера* **CHANGE\_MNG\_NO**.

Контекст выполнения *триггера* служит для того, чтобы обеспечить СУБД данными, необходимыми для корректного выполнения иницизируемого оператора *SQL*. Эти данные представляют собой набор **изменений состояния**, где каждое изменение состояния описывает изменение данных в целевой таблице *триггера*. Изменение состояния включает следующие данные:

- *триггерное* событие - **INSERT**, **UPDATE** или **DELETE** ;

- имя *предметной таблицы триггера* ;
- имена столбцов *предметной таблицы*, специфицированных в определении *триггера* (только для *триггеров* по **UPDATE** );
- набор **переходов** ( *представление* всех строк, вставляемых в *предметную таблицу*, *модифицируемых* в ней или *удаляемых* из нее), список всех *триггеров* уровня **STATEMENT**, уже *выполненных* в некотором (не обязательно активном) *контексте* выполнения, и список всех *триггеров* уровня **ROW**, уже *выполненных* в некотором (не обязательно активном) *контексте* выполнения, и строк, над которыми эти *триггеры* выполнялись.

Отслеживание уже *выполненных триггеров* ведется для предотвращения многократного *выполнения* одного и того же *триггера* в результате возникновения одного события, что могло бы потенциально привести к заикливанию *выполнения* системы *триггеров*.

При создании *контекста* *выполнения триггера* его набор изменений состояния изначально пуст. В набор изменений состояния добавляется каждое встречающееся "новое" изменение состояния, в котором не дублируются *триггерное* событие существующего изменения состояния, имя *предметной таблицы* и имена столбцов *предметной таблицы*. Набор переходов каждого изменения состояния изначально пуст, и переходы добавляются при каждом обновлении *предметной таблицы*, ассоциированной с изменением состояния (включая обновления, производимые *ссылочными действиями* ).

### Возможности использования старых и новых значений

Мы уже продемонстрировали использование старых и новых значений в определении *триггера* **DEPT\_CORRECTION\_1**. Поскольку эта возможность является важной особенностью языка SQL, обсудим ее более подробно.

Сначала немного поговорим о синтаксисе. Итак, в определении *триггера* может присутствовать раздел **REFERENCING old\_or\_new\_values\_alias\_list**, причем список определений псевдонимов может включать следующие элементы:

**OLD [ ROW ] [ AS ] correlation\_name**  
**NEW [ ROW ] [ AS ] correlation\_name**  
**OLD TABLE [ AS ] identifier**  
**NEW TABLE [ AS ] identifier**

Каждая из этих конструкций может входить в список определений псевдонимов не более одного раза, и спецификации **OLD ROW** и **NEW ROW** могут присутствовать только в определении *триггеров* уровня **ROW**. Определяемые корреляционные имена и псевдонимы можно использовать внутри *триггера* для ссылок на значения *предметной таблицы*. Если определяется корреляционное имя для новых значений ( **NEW ROW** ) или псевдоним для нового содержимого таблицы ( **NEW TABLE** ), то эти имена можно использовать для ссылок на значения, которые будут существовать в *предметной таблице* после выполнения операций **INSERT** или **UPDATE**. Если же определяется корреляционное имя для старых значений ( **OLD ROW** ) или псевдоним для старого содержимого таблицы ( **OLD TABLE** ), то данные имена можно использовать для ссылок на значения, которые существовали в *предметной таблице* до выполнения операций **UPDATE** или **DELETE**. Конечно, нельзя использовать **NEW ROW** или **NEW**

**TABLE** в триггерах **DELETE**, поскольку никакие новые значения не создаются. Аналогично, нельзя использовать **OLD ROW** или **OLD TABLE** в триггерах **INSERT**, поскольку никакие старые значения не существовали.

Таблицы, на которые указывают корреляционные имена или псевдонимы, называются *переходными*. Эти таблицы не сохраняются в базе данных долговременно; они создаются и уничтожаются динамически, по мере надобности в контексте выполнения триггера. В триггерах уровня **ROW** можно использовать корреляционное имя, определенное в конструкции **OLD ROW**, для ссылки на значения строки, удаляемой или модифицируемой иницилирующим оператором, в том виде, в котором данная строка существовала в предметной таблице до того, как была удалена или модифицирована при выполнении иницилирующего оператора. В триггерах этого уровня можно также использовать псевдоним, определенный в конструкции **OLD TABLE**, для ссылки на любое значение переходной таблицы в том виде, в котором она находилась до удаления или модификации очередной строки при выполнении иницилирующего оператора. Аналогично обстоят дела с использованием корреляционных имен и псевдонимов, определенных в конструкциях **NEW ROW** и **NEW TABLE**.

Для триггеров категории **BEFORE** имеется существенное ограничение: в них не разрешается использовать конструкции **OLD TABLE** и **NEW TABLE**, а внутритриггерный SQL-оператор не может производить какие-либо изменения в базе данных. Основанием для такого ограничения является то, что на переходные таблицы, порождаемые **OLD TABLE** и **NEW TABLE**, могут существенно влиять ссылочные действия, которые активизируются в результате изменений базы данных при выполнении внутритриггерного SQL-оператора. Поэтому значения строк в таких таблицах могут оказаться нестабильными и недостаточно предсказуемыми, если триггер срабатывает раньше действия триггерного оператора SQL.

### Обработка нескольких триггеров, связанных с одной предметной таблицей

В SQL:1999 не запрещается определение нескольких триггеров, ассоциированных с одной предметной таблицей, относящихся к одной и той же категории (**BEFORE** или **AFTER**) и срабатывающих по одному и тому же событию. Понятно, что при возникновении условия срабатывания всех таких триггеров система должна выбрать порядок, в котором они будут выполняться.

Решение, принятое в SQL, является предельно простым, хотя и несколько странным. При определении каждого триггера фиксируется временная метка выполнения оператора **CREATE TRIGGER**, и все триггеры, ассоциированные с одной предметной таблицей, относящиеся к одной и той же категории (**BEFORE** или **AFTER**) и срабатывающие по одному и тому же событию, упорядочиваются в соответствии со своими временными метками. Тогда при возникновении условия срабатывания всех триггеров одной группы сначала выполняется первый триггер, затем второй и т.д. В стандарте не специфицируется точность временной метки, связываемой с триггером, и если в одной группе обнаруживаются два или более триггеров с неразличимыми временными метками, то порядок их выполнения должен определяться в реализации.

Подход к установлению порядка выполнения триггеров в соответствии с их временными метками может вызвать чисто практические трудности у пользователей SQL-ориентированных СУБД. Например, если в ходе разработки приложения выяснится потребность в определении нового триггера, который должен выполняться раньше некоторого существующего триггера той же группы, то стандарт не может предложить

ничего лучшего, кроме как уничтожить определения всех *триггеров* этой группы, а затем заново определить их в нужном *порядке*.

И еще одно интересное свойство *триггеров* в SQL:1999. Как уже говорилось ранее в этом разделе, каждый *инициируемый SQL-оператор* должен являться *атомарным*, т. е. если его выполнение завершается неуспешно, то в базе данных не должно остаться никаких следов подобного *выполнения*. Но в стандарте говорится больше: неуспешное *выполнение* хотя бы одного *триггера* из группы с одинаковым условием срабатывания должно приводить к отмене результатов выполнения *инициируемых SQL-операторов* всех *триггеров* этой группы, а также к отмене результатов выполнения самого *инициирующего SQL-оператора* <sup>16</sup>.

## Триггеры и ссылочные действия

В разделе "Введение" лекции 12 мы достаточно подробно обсуждали механизм определения *ссылочных действий*, служащий для автоматической *поддержки ссылочной целостности*. Напомним, что *ссылочные действия* автоматически модифицируют значения *внешнего ключа* соответствующей таблицы при удалении или модификации строк таблицы, на которую указывают ссылки.

Конечно, *ссылочные действия* весьма напоминают *триггеры*, и в некоторых SQL-ориентированных СУБД они реализуются на основе общего механизма *триггеров*. Разработчики *стандарта SQL:1999* считают этот подход неудачным, поскольку процедурная природа *триггеров* входит в противоречие с тщательно разработанной декларативной основой *ссылочных ограничений целостности*. Другими словами, спецификация *ссылочной целостности*, содержащаяся в стандарте, препятствует возможности встраивания в *триггер* упрощенного процедурного кода.

Однако даже в тех СУБД, где не смешиваются механизмы *ссылочных действий* и *триггеров*, неминуемо возникает взаимосвязь между *ссылочными действиями*, изменяющими некоторую таблицу, и *триггерами*, которые определены в этой таблице или также изменяют ее. В SQL:1999 эта взаимосвязь немного упрощается за счет того, что контроль всех *ограничений целостности* (включая *ссылочные ограничения*) и выполнение всех *ссылочных действий* должны производиться до срабатывания *триггеров* категории **AFTER**. Если выполняется некоторая операция обновления таблицы **T**, то после ее выполнения и срабатывания всех *ссылочных действий* инициируются все *триггеры*, ассоциированные с таблицей **T** и видом произведенной операции, а также соответствующие *триггеры*, ассоциированные с любой таблицей, которая затрагивалась *ссылочным действием*, если в этой таблице была изменена хотя бы одна строка. Конечно, срабатывание *триггера* может привести к новым *ссылочным действиям*, которые повлекут за собой срабатывание других *триггеров* и т.д. <sup>17</sup>

В заключение этого раздела, посвященного механизму *триггеров*, заметим, что многие спецификации *стандарта SQL:1999* выглядят недостаточно убедительными. По всей видимости, полезные на практике *триггеры* слишком сложны с точки зрения теории. Создается впечатление, что за годы, прошедшие после завершения проекта System R, с подобными трудностями так и не удалось справиться. Отсюда практический совет: если вам действительно требуется использование *триггеров*, обращайтесь к документации используемой вами СУБД, а если и документация не содержит ясных рекомендаций, прибегайте к осмысленным экспериментам.

## Заключение

В этой лекции мы обсудили важные аспекты языка *SQL*, относящиеся к механизмам обновления данных. В первом разделе были рассмотрены *операторы* прямого *SQL*, предназначенные для вставки, модификации и удаления данных из существующих таблиц. *Операторы UPDATE и DELETE* этой категории иногда называют поисковыми, поскольку в них включаются условия на строки таблицы, которые должны быть модифицированы или удалены. В языке *SQL* определены также **позиционные операторы** модификации и удаления строк, а также **динамические позиционные** варианты данных операторов, но для их обсуждения требуется общее рассмотрение встраиваемого и динамического *SQL*, что выходит за рамки данного курса. На мой взгляд, поисковые версии операторов модификации и удаления хорошо характеризуют соответствующие возможности языка *SQL*. Кроме того, оператор *INSERT*, представленный в этой лекции, специфицирован в языке *SQL* только в таком варианте.

Второй раздел лекции посвящен обсуждению возможностей языка *SQL*, связанных с *применимостью операций обновления базы данных через виртуальные таблицы*, в том числе через *представления*. Мы рассмотрели ограничения языка *SQL/92*, накладываемые на *виртуальные таблицы*, к которым *применимы операции обновления*. Отмечалось, что эти ограничения являются достаточными, но не необходимыми для применения операций обновления. Был описан подход *стандарта SQL:1999*, где предлагаются рекомендации, но не требования, которых следует придерживаться реализациям *SQL*, чтобы соответствовать стандарту.

Наконец, в третьем разделе лекции рассматривался механизм *триггеров*. В первом подразделе упоминались основные понятия *триггеров*, которые были введены при выполнении проекта *System R*. Далее приводились основные синтаксические конструкции, предназначенные для определения *триггеров*, а также была описана их *базовая семантика*. В следующем подразделе обсуждались принципы *выполнения триггеров*, заложенные в стандарт *SQL:1999*. Наконец, в заключение раздела были рассмотрены имеющиеся взаимосвязи между *ссылочными действиями* и *триггерами*.

Один из основных выводов лекции состоит в том, что в *стандарте SQL:1999* спецификации многих аспектов, относящихся к обновлению баз данных, обоснованы недостаточно убедительно. В ряде случаев разработчики стандарта ожидают улучшения спецификаций в следующих версиях стандарта.

Часть следующей лекции, относящаяся к средствам языка *SQL*, которые предназначены для управления транзакциями, также имеет непосредственное *отношение* к операторам обновления баз данных.

## Лабораторная работа8. Средства языка SQL для обеспечения авторизации доступа к данным, управления транзакциями, сессиями и подключениями

### Введение

В этой лекции мы обсудим средства языка, которые касаются скорее *администраторов баз данных*, нежели конечных пользователей или программистов приложений. Но надо сказать, что любой квалифицированный *пользователь SQL-ориентированной базы данных* должен иметь *представление* об административных средствах *SQL* (тем более что средства *управления транзакциями* во многом затрагивают и его интересы).

Данная Лабораторная работа включает материал, в меньшей степени концептуально связанный, чем это было в предыдущих лекциях курса, посвященных языку SQL. В первом из основных разделов лекции мы обсудим базовые идеи *авторизации доступа к данным*, заложенные в основу языка SQL. Метод *авторизации доступа*, используемый в SQL, относится к **мандатным (mandatory) видам защиты данных**. При этом подходе с каждым зарегистрированным в системе пользователем (**субъектом**) и каждым защищаемым объектом системы связывается **мандат**, определяющий действия, которые может выполнять данный субъект над данным объектом. В отличие от такого подхода, при применении **дискреционного (discretionary) метода ограничения доступа** с каждым из объектов системы связывается одна или несколько **категорий пользователей**, каждой из которых позволяются или запрещаются некоторые действия над объектом.

Следующий раздел посвящен фундаментальному в области баз данных (и не только) понятию **транзакции** - последовательности операций над базой данных (в общем случае включающей *операции обновления базы данных*), которая воспринимается системой как одна неделимая операция. При классическом подходе к управлению транзакциями следуют принципу **ACID (Atomicity, Consistency, Isolation, Durability)**. Этому принципу следовали и разработчики языка SQL. Однако понятие транзакции выходит далеко за пределы SQL; *механизмы управления транзакциями* составляют отдельную и большую исследовательскую область. В данной лекции мы не будем углубляться в технические детали *управления транзакциями* и ограничимся возможностями, заложенными в язык SQL.

Наконец, в последнем *основном разделе* лекции мы обсудим средства языка SQL, предназначенные для управления **сессиями** и **подключениями** пользовательских приложений. Подавляющее большинство *реализаций языка SQL* основывается на архитектурной модели **клиент-сервер**. Приложения обычно выполняются на **клиентской** аппаратуре, отделенной (по крайней мере, логически) от **серверной** аппаратуры, на которой работает собственно **СУБД**. Чтобы получить *доступ* к базе данных, *приложение* должно **подключиться** к серверу и образовать **сессию** в этом подключении. У приложения может одновременно существовать несколько подключений к разным серверам баз данных, но не более одной сессии в каждом подключении.

## Поддержка авторизации доступа к данным в языке SQL

В общем случае *база данных* является слишком дорогостоящим предметом, чтобы можно было использовать ее в автономном режиме. Обычно с достаточно большой базой данных (параллельно или последовательно) работает много приложений и пользователей, и не для всех них было бы разумно обеспечивать равноправный *доступ* к хранящимся данным.

В языке SQL (SQL:1999) предусмотрены возможности *контроля доступа* к разным *объектам базы данных*, в том числе к следующим объектам:

- таблицам;
- столбцам таблиц;
- представлениям;
- доменам;
- наборам символов<sup>1</sup> ;
- порядкам сортировки символов (*collation*);
- преобразованиям (*translation*);

- *триггерам*;
- подпрограммам, вызываемым из SQL;
- определенным пользователями типам.

В совокупности в SQL:1999 может поддерживаться девять видов защиты разных объектов в соответствии со следующими возможными действиями (см. [табл.18.1](#)).

При разработке средств *контроля доступа* к объектам баз данных создатели SQL придерживались принципа *сокрытия информации* об объектах, содержащихся в *схеме базы данных*, от пользователей, которые лишены доступа к этим объектам. Другими словами, если некоторый *пользователь* не обладает, например, привилегией на просмотр таблицы *PRO*, то при выполнении операции **SELECT \* FROM PRO** он получит такое же *диагностическое сообщение*, как если бы *таблица PRO* не существовала. Если бы в случае отсутствия этой таблицы и в случае отсутствия *привилегии доступа* выдавались разные *диагностические сообщения*, то непривилегированный *пользователь* получил бы данные о том, что интересующая его *таблица* существует, но он лишен доступа к ней.

| Таблица 18.1.                         |                     |                                                                                                        |
|---------------------------------------|---------------------|--------------------------------------------------------------------------------------------------------|
| Вид защиты и соответствующее действие | Название привилегии | Применимо к следующим объектам                                                                         |
| Просмотр                              | SELECT              | Таблицы, столбцы, подпрограммы, вызываемые из SQL                                                      |
| Вставка                               | INSERT              | Таблицы, столбцы                                                                                       |
| Модификация                           | UPDATE              | Таблицы, столбцы                                                                                       |
| Удаление                              | DELETE              | Таблицы                                                                                                |
| Ссылка                                | REFERENCES          | Таблицы, столбцы                                                                                       |
| Использование                         | USAGE               | Домены, определенные пользователями типы, наборы символов, порядки сортировки символов, преобразования |
| Инициирование                         | TRIGGER             | Таблицы                                                                                                |

|              |         |                                 |
|--------------|---------|---------------------------------|
| Выполнение   | EXECUTE | Подпрограммы, вызываемые из SQL |
| Подтипизация | UNDER   | Структурные типы                |

В лекции 12 мы бегло упоминали, что в SQL-ориентированной системе каждому зарегистрированному в системе пользователю соответствует его уникальный *идентификатор* (в стандарте используется термин *идентификатор авторизации*, *authorization identifier* - *authID*). Как мы отмечали, в стандарте SQL:1999 не зафиксированы точные правила представления *идентификатора пользователя*, хотя обычно в реализациях SQL *ниладическая функция CURRENT USER* выдает текстовую строку, содержащую регистрационное имя пользователя, как оно сохраняется в файлах соответствующей операционной системы (ОС). *Привилегии доступа к объектам базы данных* могут предоставляться пользователям, представляемым своими идентификаторами, а также ролям<sup>2</sup> (см. следующий подраздел), выполнение которых, в свою очередь, может предоставляться пользователям. Кроме того, в SQL поддерживается концепция *псевдоидентификатора (или идентификатора псевдо) пользователя PUBLIC*, который соответствует любому приложению или пользователю, зарегистрированному в системе баз данных. "Пользователю" *PUBLIC* могут предоставляться *привилегии доступа к объектам базы данных*, как и любому другому пользователю.

В модели *контроля доступа SQL* создатель любого объекта базы данных автоматически становится *владельцем* этого объекта. При этом *владелец объекта* может идентифицироваться либо своим *идентификатором пользователя*, либо *именем своей роли*. Вообще говоря, *владелец объекта* обладает полным набором *привилегий* для выполнения действий над объектом (с одним исключением, которое мы обсудим в данном разделе позже). *Владелец объекта*, помимо прочего, обладает *привилегией на передачу* всех (или части) своих *привилегий* другим пользователям или ролям. В частности, *владелец объекта* может передать другим пользователям или ролям *привилегию на передачу привилегий* последующим пользователям или ролям (эти действия с *передачей привилегий на передачу привилегий* могут продолжаться рекурсивно).

Во многих реализациях поддерживаются *привилегии уровня DBA (DataBase Administrator)* для возможности выполнения операций *DDL - Data Definition Language* (*CREATE*, *ALTER* и *DROP* над объектами, входящими в схему базы данных). В стандарте SQL требуется лишь соблюдение следующих правил.

- Любые пользователь или его *роль* могут выполнять любые операции *DDL* внутри схемы, которой владеют<sup>3</sup>.
- Не допускается выполнение каких-либо операций *DDL* внутри схемы, которой не владеет пользователь или *роль*, пытающиеся выполнить соответствующую операцию.
- Эти правила не допускают исключений.

### Пользователи и роли

Как говорилось в начале этого раздела, любой пользователь характеризуется своим *идентификатором авторизации* (*authID*). В стандарте ничего не говорится о том, что *authID* должен быть идентичен *регистрационному имени пользователя* в смысле

операционной системы. Согласно стандарту SQL:1999, *authID* строится по тем же правилам, что и любой другой идентификатор, и может включать до 128 символов. Тем не менее во многих реализациях SQL, выполненных в среде ОС семейства UNIX, длина *authID* составляет не более восьми символов, как это свойственно ограничениям на длину регистрационного имени в этих ОС.

В стандарте языка SQL не специфицированы средства создания *идентификаторов авторизации*. Если говорить более точно, в стандарте не определяется какой-либо явный способ создания допустимых *идентификаторов пользователей*. *Идентификатор авторизации* может являться либо *идентификатором пользователя*, либо *именем роли*, а для создания *ролей* в SQL поддерживаются соответствующие средства (см. ниже). Но в соответствии с правилами стандарта SQL, все *authID* должны отслеживаться СУБД (имеются в виду все *authID*, для которых существует хотя бы одна *привилегия*). И в стандарте поддерживаются точные правила порождения и распространения *привилегий*. *Привилегии* по отношению к *объекту базы данных* предоставляются системой владельцу схемы при создании объекта в этой схеме, и *привилегии* могут явно передаваться от имени одного *authID* другому *authID* при наличии у первого *authID* *привилегии на передачу привилегий*.

Итак, *authID* может являться либо *идентификатором пользователя*, либо *идентификатором роли*. Попробуем разобраться в сути термина *роль*. При работе с большими базами данных в крупных организациях часто сотни служащих производят над базой данных одни и те же операции. Конечно, для этого каждый из служащих должен быть зарегистрированным пользователем соответствующей системы баз данных и тем самым, обладать собственным *authID*. Используя базовые средства *авторизации доступа* (зафиксированные в стандарте SQL/92), можно предоставить каждому пользователю группы одни и те же *привилегии доступа* к требуемым *объектам базы данных*. Но схема *авторизации доступа* при этом становится очень сложной<sup>4</sup>. В некотором смысле *имя роли* идентифицирует динамически образуемую группу пользователей системы баз данных, каждый из которых обладает, во-первых, *привилегией* на исполнение данной *роли* и, во-вторых, всеми *привилегиями* данной *роли* для доступа к *объектам базы данных*. Другими словами, наличие *ролей* упрощает построение и администрирование системы *авторизации доступа*. Проиллюстрируем это на [рис.18.1](#).

Каждая стрелка на [рис.18.1](#) соответствует мандату доступа (паре *< authID, набор\_привилегий\_доступа\_к\_объекту\_БД >*), который требуется сохранять в каталоге базы данных и проверять при попытке доступа от имени *authID*. Как видно, в случае (a) требуется сохранение и проверка *n\*m* *мандатов*, где *n* - число пользователей в группе, а *m* - число *объектов базы данных*, для которых пользователи группы должны иметь одни и те же *привилегии*. В случае (b) число требуемых для корректной работы *мандатов* равно лишь *n+m*, и схема *авторизации* резко упрощается.

Группы пользователей, объединенных одной *ролью*, являются динамическими, поскольку в SQL поддерживаются возможности предоставления пользователю *привилегии* на исполнение данной *роли* и лишения пользователя этой *привилегии* (см. ниже в этом разделе). Более того, имеются возможности предоставления заданной *роли* 1 всех или части *привилегий* другой *роли* 1. Естественно, что при этом *привилегии* изменяются у всех пользователей, которые могут исполнять *роль* 1.

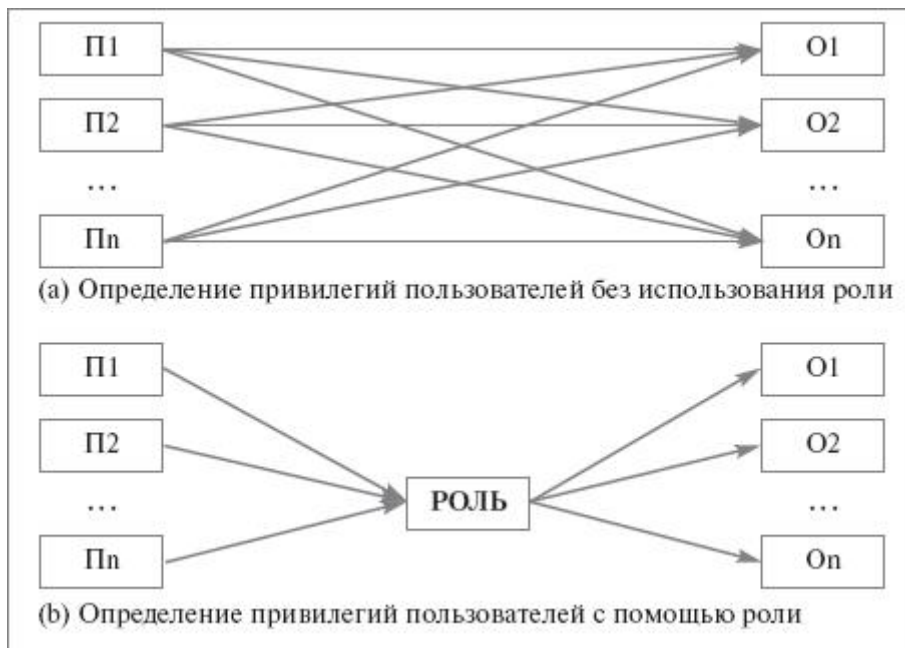


**Рис. 18.1.** Привилегии, пользователи и роли

Более того, имеются возможности предоставления заданной роли **A** всех или части привилегий другой роли **B**. Естественно, что при этом привилегии изменяются у всех пользователей, которые могут исполнять роль **A**.

### Применение идентификаторов пользователей и имен ролей

В этом подразделе мы вынуждены использовать понятие *SQL-сессии*, которое более последовательно обсуждается в третьем основном разделе лекции. Как и в предыдущих лекциях, посвященных языку SQL, мы можем оправдать подобную нелогичность только рекурсивной природой стандарта SQL<sup>5</sup>.

Итак, в любой момент заданная *SQL-сессия* ассоциируется с идентификатором пользователя, называемым идентификатором пользователя *SQL-сессии* и с именем роли, называемым именем роли *SQL-сессии*. Почти всегда привилегии, связанные с этими идентификатором и именем, используются для определения допустимости выполнения различных операций во время данной сессии.

В стандарте не специфицированы все способы ассоциирования *authID* с *SQL-сессией*. Определено лишь то, что если сессия образуется с помощью оператора *CONNECT* (см. раздел "Подключения и сессии"), то *authID* указывается в качестве параметра соответствующей операции. Для реализаций SQL допускается, чтобы пользовательский идентификатор *SQL-сессии* совпадал с регистрационным именем пользователя с точки зрения операционной системы или являлся идентификатором, специально устанавливаемым специалистами организации, ответственными за обеспечение безопасности. Кроме того, допускается наличие в реализации оператора *SET SESSION AUTHORIZATION*, применение которого приводит к смене пользовательского идентификатора *SQL-сессии*. В начале *SQL-сессии* значение текущего идентификатора пользователя *SQL-сессии* совпадает со значением пользовательского идентификатора *SQL-сессии*, и такая ситуация сохраняется до тех пор, пока пользовательский идентификатор *SQL-сессии* не будет каким-либо образом изменен. Значение текущего

*пользовательского* *идентификатора* *SQL-сессии* возвращается вызовом *ниладической* функции **SESSION\_USER** (Лабораторная работа13).

Для каждой *SQL-сессии* существует также *текущее имя роли* (это имя можно получить путем вызова функции **CURRENT\_ROLE**). Сразу после образования сессии *текущему имени роли* соответствует *неопределенное значение*, что трактуется как "*роль для сессии не назначена*". Имеется несколько способов подмены *пользовательского идентификатора* и/или *имени роли SQL-сессии*. При этом если задается *идентификатор пользователя*, то одновременно полагается, что неявно указывается *имя роли*, имеющее *неопределенное значение*. Если же задается *имя роли*, то, за несколькими исключениями, считается, что неявно указывается *идентификатор пользователя*, имеющий *неопределенное значение*. Более подробно мы обсудим подобные возможности ниже в этом разделе.

Если либо *текущий идентификатор пользователя*, либо *текущее имя роли* содержат *неопределенное значение*, то тот идентификатор или то имя, у которого значение не является *неопределенным*, используется в качестве текущего **authID** *SQL-сессии*. Если ни *текущий идентификатор*, ни *текущее имя* не содержат **NULL**, то текущим **authID** сессии служит *текущий идентификатор пользователя*.

Как и везде в этом курсе, мы опустим детали, относящиеся к особенностям *авторизации* при использовании *встраиваемого* и *динамического SQL*.

### Создание и ликвидация ролей

Для создания новой *роли* используется оператор **CREATE ROLE**, определяемый следующим синтаксическим правилом:

```
CREATE ROLE role_name
[WITH ADMIN { CURRENT_USER | CURRENT_ROLE }]
```

Имя создаваемой *роли* должно отличаться от любого *идентификатора авторизации*, уже определенного и сохраненного в базе данных. В случае успешного создания *роли* некоторый **authID** получает *привилегию* на исполнение данной *роли*. Если в операторе **CREATE ROLE** не содержится раздел **WITH ADMIN**, то *привилегию* на исполнение *роли* получает *текущий идентификатор пользователя SQL-сессии*, если значение этого идентификатора отлично от **NULL**; иначе *привилегия* на исполнение *роли* дается *текущему имени роли сессии*.

Если в состав оператора включается раздел **WITH ADMIN**, то можно выбрать, будет ли являться владельцем *роли* **authID**, соответствующий *текущему идентификатору пользователя SQL-сессии*, или **authID**, соответствующий *текущему имени роли* (при условии, что соответствующие *текущий идентификатор* или *текущее имя* не содержат **NULL**). Кроме того, включение этого раздела означает, что **authID** - *владелец роли* получает право на *передачу привилегии* исполнения данной *роли* другим **authID**.

В соответствии со стандартом SQL:1999, *привилегии*, требуемые для выполнения оператора **CREATE ROLE**, определяются в реализациях SQL. Например, в некоторых реализациях выполнение этой операции разрешается только *администратору базы данных*.

Существующую *роль* можно ликвидировать с помощью оператора

***DROP ROLE role\_name***

Для выполнения этой операции требуется, чтобы текущий ***authID*** *SQL-сессии* прямо или косвенно (через цепочку *ролей*) являлся владельцем ликвидируемой *роли*. При ликвидации *роли*, прежде всего, изымается *привилегия* на ее исполнение у всех ***authID***, которым данная *привилегия* была ранее передана.

### Передача привилегий и ролей

Для *передачи* *привилегий* и *ролей* от одних ***authID*** другим поддерживается оператор ***GRANT***, который мы обсудим отдельно для случаев *передачи привилегий* и *передачи ролей*.

### Передача привилегий

В случае *передачи привилегий* используется следующий синтаксис оператора ***GRANT***:

```
GRANT { ALL PRIVILEGES | privilege_commalist }
 ON privilege_object
TO { PUBLIC | authID_commalist } [WITH GRANT OPTION]
 [GRANTED BY { CURRENT_USER | CURRENT_ROLE }]
privilege ::= SELECT [column_name_commalist]
 | DELETE
 | INSERT [column_name_commalist]
 | UPDATE [column_name_commalist]
 | REFERENCES [column_name_commalist]
 | USAGE
 | TRIGGER
 | EXECUTE
privilege_object ::= [TABLE] table_name
 | DOMAIN domain_name
 | CHARACTER SET character_set_name
 | COLLATION collation_name
 | TRANSLATION translation_name
```

Поскольку ***authID*** может являться *идентификатором пользователя* или *именем роли*, *привилегии* могут передаваться от пользователей пользователям, от пользователей *ролям*, от *ролей ролям* и от *ролей* пользователям.

В списке *привилегий* можно использовать ***SELECT***, ***DELETE***, ***INSERT***, ***UPDATE***, ***REFERENCES*** и ***TRIGGER*** только в том случае, когда в качестве объекта *привилегий* указывается таблица. Соответственно, список *привилегий* может состоять из единственной *привилегии* ***USAGE*** только в том случае, когда объектом является домен, набор символов, порядок сортировки или *трансляция*. Если в списке *привилегий* указывается более одной *привилегии*, то они все передаются указанным ***authID***, но для этого текущий ***authID*** *SQL-сессии* должен обладать *привилегией* на *передачу привилегий*.

Использование ключевого слова **ALL PRIVILEGES** вместо явного задания списка *привилегий* означает, что передаются все *привилегии* доступа к соответствующему объекту базы данных, которыми обладает текущий **authID** SQL-сессии.

Как показывает синтаксис, один оператор **GRANT** позволяет передавать *привилегии* доступа только к одному объекту, но в том случае, когда объектом является таблица, разные *привилегии* могут передаваться по отношению к одному и тому же набору столбцов или к разным наборам. Если при указании *привилегий* **SELECT**, **DELETE**, **UPDATE** и **REFERENCES** список имен столбцов не задается, передаются *привилегии* по отношению ко всем столбцам таблицы. Заметим, что эти *привилегии* касаются всех существующих столбцов данной таблицы, а также всех столбцов, которые когда-либо будут к ней добавлены.

Включение в оператор необязательного раздела **WITH GRANT OPTION** означает, что получателям передаваемых *привилегий* дается также *привилегия* на дальнейшую *передачу* полученных *привилегий*, включая *привилегию* на *передачу привилегий*. Включение в оператор раздела **GRANTED BY** позволяет явно указать, передаются ли *привилегии* от имени текущего идентификатора пользователя или же *текущего имени роли*.

При проверке возможности выполнения операции в SQL-сессии учитываются *привилегии* текущего **authID** SQL-сессии, а также *привилегии* всех *ролей*, которые переданы данному **authID**. Поскольку этим *ролям* могли быть переданы другие *роли*, обладающие собственными *привилегиями*, анализ возможности выполнения операции является рекурсивной процедурой.

Если одна и та же *привилегия* передается более одного раза одному и тому же **authID2** от имени одного и того же **authID1**, то возникает ситуация, называемая избыточной дублирующей *привилегией*. Эта ситуация не вызывает дополнительных проблем, поскольку избыточная *передача привилегии* игнорируется. Для аннулирования данной *привилегии* у **authID2** от имени **authID2** требуется выполнение всего лишь одной операции **REVOKE** (см. ниже в этом разделе). Если *привилегия* была один раз передана **authID2** от имени **authID1** вместе с *привилегией* на *передачу этой привилегии* (**WITH GRANT OPTION**), а в другой раз - без этой опции (порядок действий не является существенным), то **authID2** обладает данной *привилегией* и *привилегией* на ее *передачу*.

Если предпринимается попытка *передачи* нескольких *привилегий*, но соответствующий **authID** не обладает ни одной из них, то фиксируется ошибка. Аналогично, если производится попытка *передачи* нескольких *привилегий* с *передачей привилегии* на *передачу привилегий*, но соответствующий **authID** не обладает *привилегией WITH GRANT OPTION* ни для одной из передаваемых *привилегий*, то фиксируется ошибка. Наконец, если производится попытка *передачи* нескольких *привилегий* с *передачей привилегии* на *передачу привилегий* и соответствующий **authID** обладает *привилегией* на *передачу* только части этих *привилегий*, то в результате выполнения операции вырабатывается предупреждение, но соответствующая часть *привилегий* передается с *привилегией WITH GRANT OPTION*.

## Привилегии и представления

При определении представлений действуют специальные *правила определения привилегий над этими представлениями*. Если при создании обычных объектов базы данных, таких,

как таблица или домен, текущий *authID* автоматически получает все возможные *привилегии* доступа к соответствующему объекту, включая *привилегию* на передачу *привилегий*, то для представлений ситуация иная. Поскольку создаваемое представление всегда основывается на одной или нескольких базовых таблицах (или представлениях), *привилегии*, которые получает создатель представления, должны основываться на *привилегиях*, которыми располагает текущий *authID* по отношению к этим базовым таблицам или представлениям.

Например, чтобы операция создания представления была выполнена успешно, текущий *authID* должен обладать *привилегией* **SELECT** по отношению ко всем базовым таблицам и представлениям, на которых основывается новое представление. Тогда текущий *authID* автоматически получит *привилегию* **SELECT** для нового представления. Но текущий *authID* сможет передавать эту привилегию другим *authID* только тогда, когда обладает соответствующей *привилегией* для всех базовых таблиц и представлений, на которых основывается новое представление. Аналогичным образом на представление распространяются *привилегии* **DELETE**, **INSERT**, **UPDATE** и **REFERENCES**. Поскольку *триггеры* над представлениями создавать не разрешается, *привилегия* **TRIGGER** представлениям не передается.

Наконец, посмотрим, что происходит при смене *привилегий* владельца представления по отношению к таблицам, на которых основано это представление. Для простоты предположим, что представление **V** основано на базовой таблице **T**. Если во время создания **V** текущий *authID* (будущий владелец представления) обладал по отношению к **T** *привилегиями* **SELECT** и **INSERT**, то он будет обладать этими *привилегиями* и по отношению к **V**<sup>6</sup>. Если впоследствии владелец представления получит по отношению к **T** дополнительные *привилегии*, то он (и все *authID*, которым передавались все *привилегии* - **ALL PRIVILEGES** для **V**) получит те же *привилегии* для **V**. Должно быть понятно, каким образом обобщается этот подход на случай, когда представление определяется над несколькими таблицами или представлениями.

## Передача ролей

Для передачи ролей используется следующий вариант оператора *GRANT*:

```
GRANT role_name_commalist
TO { PUBLIC | authID_commalist } [WITH ADMIN OPTION]
[GRANTED BY { CURRENT_USER | CURRENT_ROLE }]
```

Как показывает синтаксис, оператор позволяет передавать произвольное число ролей произвольному числу *authID* (которые могут представлять собой идентификаторы пользователей или имена ролей). Как и в случае передачи *привилегий*, от данного *authID* можно передавать только те роли, которые были получены этим *authID* с *привилегией* на дальнейшую передачу (**WITH ADMIN OPTION**). При включении в состав оператора *GRANT* раздела **GRANTED BY** можно явно указать, что роли передаются от имени текущего идентификатора пользователя или же текущего имени роли.

## Изменение текущих идентификаторов пользователей и имен ролей

Как мы отмечали ранее в этом разделе, в SQL:1999 специфицированы некоторые операторы, позволяющие изменять текущий *идентификатор пользователя* и *текущее имя роли SQL-сессии*.

## Оператор SET SESSION AUTHORIZATION

Для изменения *текущего идентификатора пользователя SQL-сессии* может использоваться оператор

**SET SESSION AUTHORIZATION value\_specification**

Как указывалось в лекции 13, **value\_specification** может быть либо литералом (в данном случае литералом *типа символьных строк*), либо вызовом **ниладической** функции, такой, как **CURRENT\_USER**, **SESSION\_USER** и т. д. Если указанная спецификация значения не соответствует требованиям, предъявляемым в реализации к представлению *идентификатора пользователя*, операция изменения *текущего идентификатора пользователя* аварийно завершается.

В стандарте также говорится, что если спецификация значения, заданная в операции, формально соответствует требованиям, предъявляемым к формату *идентификатора пользователя* конкретной системы, но в действительности не представляет **известный системе идентификатор пользователя**, то опять же фиксируется ошибка, и операция не выполняется. Допускается, чтобы в реализации принималось решение о смене *идентификатора пользователя сессии* одновременно с регистрацией нового *идентификатора пользователя*. Ограничения на регистрацию таким способом нового пользователя тоже определяются на уровне реализации. После успешного выполнения оператора **SET SESSION AUTHORIZATION** *текущее имя роли* соответствующей сессии принимает значение **NULL**, так что текущим **authID** этой сессии становится заданное значение *идентификатора пользователя*.

Опять по необходимости забегаая вперед, заметим, что операцию смены *текущего идентификатора пользователя SQL-сессии* не разрешается выполнять внутри какой-либо транзакции этой сессии. Иначе терялся бы смысл *привилегий доступа*, которыми руководствуется система при выполнении операций внутри транзакции.

## Оператор SET ROLE

Для смены *текущего имени роли SQL-сессии* можно использовать оператор

**SET ROLE { value\_specification | NONE }**

Ограничения на выполнение операции **SET ROLE** почти совпадают с определенными в стандарте ограничениями на выполнение операции **SET SESSION AUTHORIZATION**. Наиболее важные отличия состоят в том, что эту операцию от имени текущего **authID** сессии всегда разрешается выполнять для *ролей*, которые переданы "пользователю" **PUBLIC** или данному текущему **authID**, а также в том, что всегда разрешается применение конструкции **SET ROLE NONE**. Выполнение последней конструкции приводит к тому, что значение *текущего имени роли* сессии становится неопределенным.

Заметим, что при смене *текущего имени роли SQL-сессии* значение *текущего пользовательского идентификатора сессии* не меняется, так что вполне вероятно, что

после выполнения операции и текущий идентификатор, и текущее *имя роли* будут иметь значения, отличные от неопределенного значения. И конечно, операция **SET ROLE NONE** будет выполнена успешно только в том случае, когда значение текущего *пользовательского идентификатора* не является неопределенным<sup>7</sup>.

## Аннулирование привилегий и ролей

Если от имени некоторого *authID* некоторые *привилегии* или *роли* были переданы одному или нескольким другим *authID*, то впоследствии первый *authID* (в сессии, где этот *authID* является текущим) можно изъять, или **аннулировать**, переданные *привилегии* или *роли* путем применения оператора **REVOKE**. Как и в случае *передачи привилегий и ролей*, способы *аннулирования привилегий и ролей* похожи, но между ними имеются некоторые отличия. Поэтому мы снова обсудим эти способы в отдельности.

### Аннулирование привилегий

Для *аннулирования привилегий* используется оператор **REVOKE**, определяемый следующим синтаксическим правилом:

```
REVOKE [GRANT OPTION FOR] privilege_commalist
ON privilege_object
FROM { PUBLIC | authID_commalist }
[GRANTED BY { CURRENT_USER | CURRENT_ROLE }]
{ RESTRICT | CASCADE }
```

Синтаксис конструкций *privilege* и *privilege\_object* такой же, как для оператора **GRANT**. Общий смысл операции должен быть понятен из синтаксиса: у указанных *authID* аннулируются указанные *привилегии доступа* к указанному *объекту базы данных*.

Первой важной особенностью оператора *аннулирования привилегий* является обязательность указания одного из ключевых слов **RESTRICT** или **CASCADE**. Если в операторе содержится **RESTRICT**, то при выполнении операции система проверит, не передавалась ли какая-либо из указанных *привилегий* каким-либо *authID* от того *authID*, у которого *привилегия* должна быть *аннулирована* (это вполне возможно, если ранее *привилегия* была передана с правом передачи). Если это действительно так, операция не выполняется; в противном случае указанные *привилегии* у указанных *authID* аннулируются. Иначе говоря, при наличии ключевого слова **RESTRICT** не допускается, например, ситуация, показанная на [рис.18.2](#).

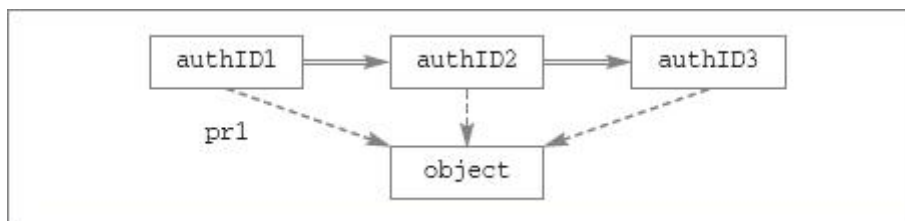


Рис. 18.2. Передача полученной привилегии

На этом рисунке **authID1** является владельцем объекта базы данных с именем **object** и, следовательно, обладает всеми привилегиями над этим объектом. Пунктирной стрелкой обозначена одна из подобных привилегий **pr1**. От имени **authID1** привилегия **pr1** была передана **authID2** вместе с привилегией на ее дальнейшую передачу. Наконец, от имени **authID2** привилегия **pr1** была передана **authID3**. Тогда операция аннулирования этой привилегии от имени **authID1** у **authID2** при наличии ключевого слова **RESTRICT** не будет выполнена успешно.

В той же ситуации привилегия была бы аннулирована для **authID2** (и для **authID3**), если бы в операторе **GRANT** присутствовало ключевое слово **CASCADE**. В общем случае если выполняется операция **REVOKE... CASCADE**, то указанные привилегии аннулируются у всех **authID**, прямо или косвенно (через промежуточные **authID**) получивших привилегии от текущего **authID** SQL-сессии, в которой выполняется данная операция.

Если в операторе содержится раздел **GRANT OPTION FOR**, но имеется ключевое слово **RESTRICT**, то указанные привилегии для указанных **authID** не аннулируются, но у указанных **authID** аннулируется привилегия передачи данных привилегий (операция должна успешно выполняться только при соблюдении обсуждавшихся ранее условий). Однако если в операторе одновременно содержатся и **GRANT OPTION FOR**, и **CASCADE**, то указанные привилегии аннулируются у всех **authID**, которые прямо или косвенно (через промежуточные **authID**) получили привилегии от текущего **authID** SQL-сессии, в которой выполняется данная операция.

Задание в операторе необязательного раздела **GRANTED BY** позволяет явно указать, что должно использоваться в качестве текущего **authID** - текущий пользовательский идентификатор или текущее имя роли SQL-сессии. Если раздел **GRANTED BY** в операторе **REVOKE** не содержится, то действия производятся от имени текущего **authID** SQL-сессии (о том, как он определяется, см. выше).

Если текущий (или указанный) **authID** не обладает ни одной из указанных в операторе **REVOKE** привилегий, то выполнение операции не производится (фиксируется ошибка). Если **authID** обладает некоторыми, но не всеми, привилегиями из числа указанных, то операция выполняется по отношению к этим некоторым привилегиям, но выдается предупреждение.

Возможны ситуации, когда у некоторого **authID** остается некоторая привилегия после выполнения операции аннулирования у этого **authID** этой привилегии. Одна из таких ситуаций проиллюстрирована на [рис.18.3](#).

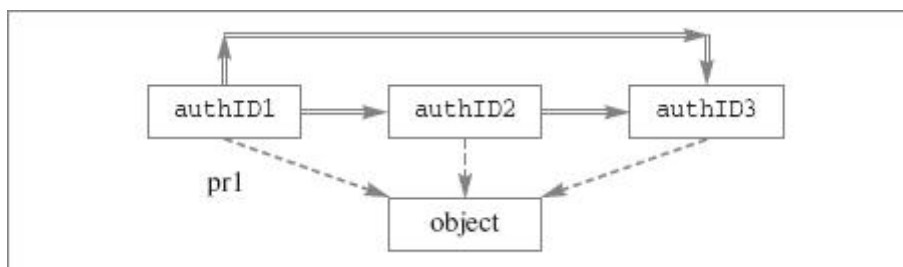


**Рис. 18.3.** Косвенная и прямая передача привилегий

Здесь *привилегия pr1* передана от *authID1* к *authID2* вместе с правом на дальнейшую *передачу* этой *привилегии*. Далее, *привилегия pr1* передается от *authID2* к *authID3*. И затем выполняется *прямая передача привилегии* от *authID1* к *authID3* (на самом деле, порядок таких действий не является существенным). Теперь предположим, что от имени *authID1* выполняется операция

**REVOKE pr1 ON object FROM authID2 CASCADED**

В соответствии с правилами SQL:1999 после выполнения этой операции *authID3* будет продолжать владеть *привилегией pr1* по отношению к объекту *object*, поскольку получил данную *привилегию двумя разными способами*. Грубо говоря, операция **REVOKE**, выполняемая от имени *authID1*, выполняется только по тем путям графа *идентификаторов авторизации* и *объектов базы данных*, которые начинаются с узлов, соответствующих *authID*, указанных в разделе **FROM** этой операции.

Далее, напомним, что если при *передаче* от *authID1* к *authID2* *привилегии* на выполнение некоторых действий над некоторой таблицей **T** (например, **UPDATE**) явно не указывается список имен столбцов этой таблицы, то *привилегия* распространяется на все столбцы этой таблицы (включая столбцы, которые, возможно, еще будут созданы). Если действительно использовался такой *способ передачи привилегий*, то в дальнейшем можно аннулировать *привилегию authID2* на модификацию отдельных (уже *определенных*) *столбцов* таблицы **T**, оставив *привилегию* на модификацию всех остальных столбцов (включая те, которые еще не созданы).

И последнее замечание. Если некоторая *привилегия* была передана *псевдоauthID PUBLIC*, то, конечно, этой *привилегией* обладают все *authID*. Но нет возможности аннулировать такую *привилегию* у отдельно указываемого *authID*. *Привилегия* была передана всем, и аннулировать ее можно только сразу у всех.

### Аннулирование ролей

Вариант оператора **REVOKE**, используемый для *аннулирования ролей*, выглядит следующим образом:

```
REVOKE [ADMIN OPTION FOR] role_name_commalist
FROM { PUBLIC | authID_commalist }
[GRANTED BY { CURRENT_USER | CURRENT_ROLE }]
{ RESTRICT | CASCADE }
```

Действие операции *аннулирования ролей* очень похоже на действие операции *аннулирования привилегий*. Отличие состоит в том, что аннулируются не *привилегии*, а *роли*, а также в том, что для *аннулирования привилегий* на *передачу роли* используется раздел **ADMIN OPTION FOR**<sup>8</sup>.

### Управление транзакциями в SQL

Область организации транзакций и управления ими настолько широка, что заслуживает отдельных книг и учебных курсов. В этом курсе мы ограничимся рассмотрением вопросов *управления транзакциями* в пределах *стандарта языка SQL*.

### ACID-транзакция

В SQL:1999 поддерживается классическое понимание транзакции, характеризуемое аббревиатурой **ACID** (от *Atomicity*, *Consistency*, *Isolation* и *Durability*). В соответствии с этим понятием под транзакцией понимается последовательность операций (например, над базой данных), обладающая следующими свойствами.

- **Атомарность** (*Atomicity*). Это свойство означает, что результаты всех операций, успешно выполненных в пределах транзакции, должны быть отражены в состоянии базы данных, либо в состоянии базы данных не должно быть отражено действие ни одной операции (конечно, здесь речь идет об операциях, изменяющих состояние базы данных). Свойство *атомарности*, которое часто называют свойством "все или ничего", позволяет относиться к транзакции, как к динамически образуемой составной операции над базой данных<sup>9</sup>.
- **Согласованность** (*Consistency*). В классическом смысле это свойство означает, что транзакция может быть успешно завершена с **фиксацией** результатов своих операций только в том случае, когда действия операций не нарушают целостность базы данных, т. е. удовлетворяют набору ограничений целостности, определенных для этой базы данных. В стандарте SQL:1999 это свойство расширяется тем, что во время выполнения транзакции разрешается устанавливать точки *согласованности* (см. ниже про точки сохранения) и явным образом проверять ограничения целостности<sup>10</sup>.
- **Изоляция** (*Isolation*). Требуется, чтобы две одновременно выполняемые транзакции<sup>11</sup> никоим образом не действовали одна на другую. Другими словами, результаты выполнения операций транзакции **T1** не должны быть видны никакой другой транзакции **T2** до тех пор, пока транзакция **T1** не завершится успешным образом.
- **Долговечность** (*Durability*). После успешного завершения транзакции все изменения, которые были внесены в состояние базы данных операциями этой транзакции, должны гарантированно сохраняться, даже в случае сбоя аппаратуры или программного обеспечения.

### Порождение транзакций в SQL:1999

В соответствии со стандартом языка SQL:1999 транзакции<sup>12</sup> могут образовываться явным образом с использованием оператора **START TRANSACTION**, либо неявно, когда выполняется оператор, для которого требуется контекст транзакции, а этого контекста не существует. Например, операторы **SELECT**, **UPDATE** или **CREATE TABLE** могут выполняться только в контексте транзакции, а для выполнения оператора **CONNECT** (см. раздел "Подключения и сессии") такой контекст не требуется, и выполнение оператора **CONNECT** не приводит к неявному образованию транзакции. Для завершения транзакции должен быть явно использован один из двух операторов - **COMMIT** (требование завершить транзакцию с фиксацией ее результатов) или **ROLLBACK** (требование завершить транзакцию с удалением результатов всех выполненных операций из состояния базы данных<sup>13</sup>).

### Установка характеристик транзакции

У каждой выполняемой транзакции имеются три характеристики, значения которых существенно влияют на действия системы при управлении транзакцией, - *уровень изоляции (isolation level)*, *режим доступа (access mode)* и *размер области диагностики*. При неявном образовании транзакции эти характеристики устанавливаются по умолчанию: транзакция получает максимальный *уровень изоляции* от одновременно выполняемых транзакций; *режим доступа*, позволяющий выполнять и операции выборки, и операции обновления базы данных; и назначаемый по умолчанию *размер области диагностики*.

Если значения характеристик транзакции, устанавливаемых по умолчанию, в некотором случае не являются пригодными, то до выполнения оператора, неявно иницилирующего транзакцию, можно явно установить характеристики данной транзакции с использованием оператора **SET TRANSACTION**. Этот оператор определяется следующими синтаксическими правилами:

```
SET TRANSACTION mode_commalist
mode ::= isolation_level
 | access_mode
 | diagnostics_size
isolation_level ::= READ UNCOMMITTED
 | READ COMMITTED
 | REPEATABLE READ
 | SERIALIZABLE
access_mode ::= READ ONLY
 | READ WRITE
diagnostics_size ::= DIAGNOSTIC SIZE value_specification
```

Операцию установки характеристик транзакции нельзя выполнять в контексте какой-либо *активной транзакции*. Выполнение операции допустимо только до образования первой транзакции *SQL-сессии* или между последовательно выполняемыми транзакциями этой сессии. В одном операторе **SET TRANSACTION** можно задать только по одному значению каждой из трех характеристик, но допускается последовательное выполнение нескольких таких операций с разными операндами.

Как видно из синтаксических правил, у характеристики *режим доступа* может быть указано одно из двух значений - **READ ONLY** или **READ WRITE**. Если устанавливается режим **READ ONLY**, то в транзакции нельзя будет выполнять никакие операции, изменяющие базу данных, в том числе операции обновления таблиц и определения новых *объектов базы данных*. Если *режим доступа* явно не указывается, по умолчанию принимается характеристика **READ WRITE**, если только в качестве значения характеристики *уровень изоляции* не указывается **READ UNCOMMITTED** (в этом случае устанавливается *режим доступа READ ONLY*).

Если указывается *размер области диагностики*, то после ключевых слов **DIAGNOSTIC SIZE** должен следовать целочисленный литерал, определяющий число диагностических элементов, которые должны разместиться в области диагностики (число *исключительных ситуаций*, предупреждений, сообщений об отсутствии данных и об успешном выполнении, которые будут вырабатываться при выполнении операторов внутри будущей транзакции). Если *размер области диагностики* явно не указывается, то решение о *размере этой области* принимается в реализации<sup>14</sup>.

*Уровни изоляции* будут подробно обсуждаться ниже, но здесь мы заметим, что если значение *уровня изоляции* явно не задано, то по умолчанию принимается *уровень изоляции SERIALIZABLE*. Кроме того, еще раз обратим внимание читателей, что одновременное задание уровня изоляции **READ UNCOMMITTED** и *режима доступа READ WRITE* не допускается.

Еще одна интересная деталь оператора установки характеристик транзакции состоит в том, что выполнение каждого следующего оператора **SET TRANSACTION** полностью перекрывает эффект выполнения предыдущего такого оператора. В частности, если в

предыдущем операторе явно задавалось значение некоторой характеристики, а в следующем это значение принимается по умолчанию, то именно значение по умолчанию будет являться значением характеристики транзакции.

### Явная инициация транзакции

Для явного образования транзакции поддерживается оператор **START TRANSACTION**, определяемый следующими синтаксическими правилами:

**START TRANSACTION mode\_commalist**

Этот оператор очень похож на **SET TRANSACTION**. Единственное (хотя и очень существенное) отличие состоит в том, что выполнение оператора **START TRANSACTION** приводит не только к установке характеристик транзакции, но и к реальной инициации транзакции.

### Уровни изоляции SQL-транзакции

В стандарте *SQL:1999* уровни изоляции определяются на основе нескольких феноменов, которые могут возникать при выполнении транзакций<sup>15</sup>.

#### Феномен "грязного" чтения (dirty read)

Этому феномену подвержены транзакции, в которых допускается возможность видеть изменения объектов базы данных, производимые другими одновременно выполняемыми и еще не зафиксированными транзакциями. Простой пример феномена "грязного" чтения показан на [рис.18.4](#).

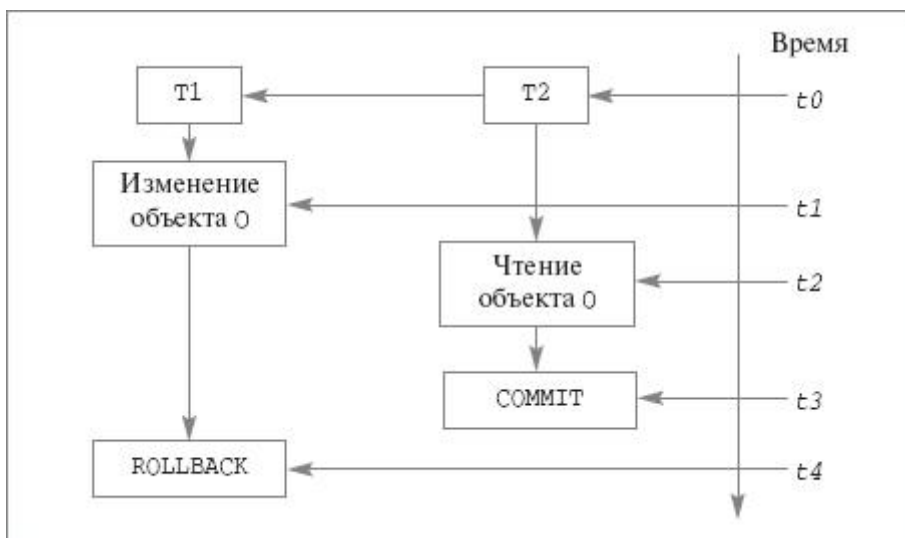


Рис. 18.4. Феномен "грязного" чтения

На этом рисунке показано, что в момент времени  $t_0$  были образованы две транзакции T1 и T2. В момент времени  $t_1$  транзакция T1 успешно выполняет операцию модификации некоторого объекта базы данных O. В момент времени  $t_2$  ( $t_2 > t_1$ ) транзакция T2 читает объект O, после чего успешно завершается в момент времени  $t_3$ . Транзакция же T1 завершается в момент времени  $t_4$  ( $t_4 > t_3$ ), причем в ней выполняется оператор **ROLLBACK**, что приводит к ликвидации в базе данных последствий изменения

объекта **O**. В результате оказывается, что в транзакции **T2** обрабатывались данные, которые реально не существуют в базе данных (отсюда и термин "грязные" данные).

В SQL феномен "грязного" чтения может наблюдаться у транзакций, выполняемых на уровне изоляции **READ UNCOMMITTED**. Рекомендуется использовать этот уровень изоляции только в тех транзакциях, для выполнения функций которых точные данные не обязательны (например, в транзакциях, производящих статистическую обработку).

### Феномен неповторяемого чтения (unrepeatable read)

Этому феномену подвержены транзакции, читающие некоторые объекты базы данных и допускающие изменения уже прочитанных объектов другими транзакциями. Пример феномена неповторяемого чтения показан на [рис.18.5](#).

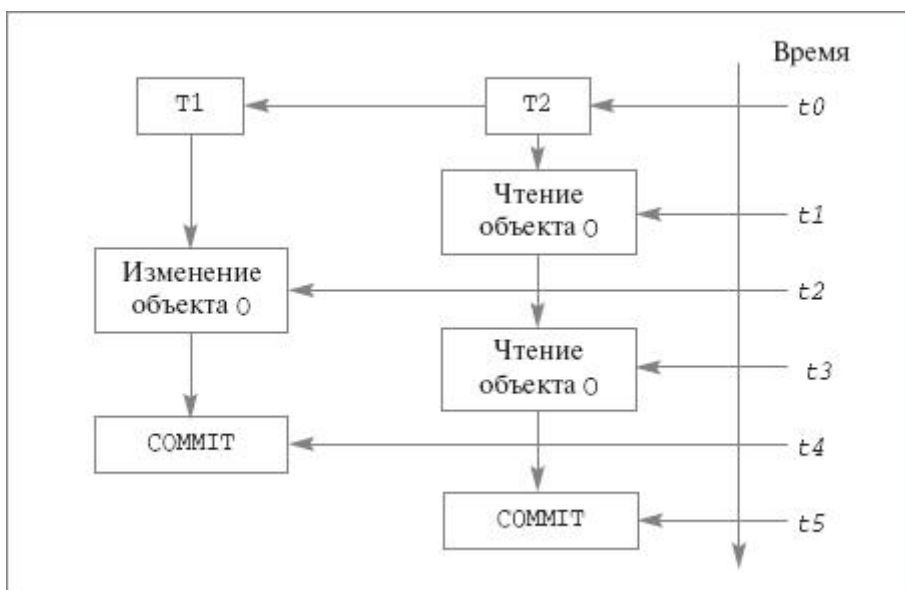


Рис. 18.5. Феномен неповторяемого чтения

На этом рисунке показано, что в момент времени **t0** были образованы две транзакции **T1** и **T2**. В момент времени **t1** транзакция **T2** выполняет операцию чтения некоторого объекта базы данных **O** (например, производит выборку строки из таблицы с указанием значения первичного ключа). В момент времени **t2** ( $t2 > t1$ ) транзакция **T1** изменяет объект **O** (модифицирует или даже удаляет). В момент времени **t3** ( $t3 > t2$ ) транзакция **T2** повторно считывает объект **O** и обнаруживает, что он изменился или вовсе отсутствует. Другими словами, в транзакции **T2** повторное выполнение выборки объекта базы данных **O** дало результат, отличный от результата первого выполнения (отсюда и происходит термин "неповторяемое чтение").

В SQL феномен неповторяемого чтения может наблюдаться у транзакций, выполняемых на уровне изоляции **READ COMMITTED** (этот уровень изоляции, как показывает его название, гарантирует отсутствие феномена "грязного" чтения).

### Феномен фантомов

Этому феномену подвержены транзакции, производящие выборку строк и таблиц базы данных и допускающие добавление к данным таблицам другими транзакциями строк,

которые удовлетворяют условию выборки. Пример *феномена фантомов* показан на [рис.18.6](#).



**Рис. 18.6.** Феномен фантомов

На этом рисунке показано, что в момент времени  $t_0$  были образованы две транзакции, **T1** и **T2**. В момент времени  $t_1$  транзакция **T2** выполняет операцию выборки строк из таблицы **R** по условию **c**. В момент времени  $t_2$  ( $t_2 > t_1$ ) транзакция **T1** выполняет над таблицей **R** операцию обновления (вставки или модификации строк), в результате которой в таблице **R** появляются дополнительные строки, удовлетворяющие условию **c**. В момент времени  $t_3$  ( $t_3 > t_2$ ) транзакция **T2** повторно выполняет операцию выборки строк из таблицы **R** по условию **c** и обнаруживает наличие в результате дополнительных фантомных строк.

В SQL феномен фантомов может наблюдаться у транзакций, выполняемых на уровне изоляции **REPEATABLE READ** (этот уровень изоляции, как показывает его название, гарантирует отсутствие феномена неповторяемого чтения).

Наконец, для транзакций, выполняемых на уровне изоляции **SERIALIZABLE**, невозможно и проявление феномена фантомов. Термин *serializable* (**сериализуемый**) используется по той причине, что при работе на данном уровне изоляции суммарный эффект выполнения набора транзакций  $\{T_1, T_2, \dots, T_n\}$  идентичен эффекту некоторого последовательного выполнения этих транзакций. Это означает предельную изолированность транзакций. Общая картина взаимосвязи уровней изоляции и феноменов транзакций показана в [таблице 18.2](#).

Таблица 18.2. Уровни изоляции и феномены

| Уровень                 | "Грязное" чтение | Неповторяемое чтение | Фантомы  |
|-------------------------|------------------|----------------------|----------|
| <b>READ UNCOMMITTED</b> | Возможно         | Возможно             | Возможны |

|                        |            |            |            |
|------------------------|------------|------------|------------|
| <i>READ COMMITTED</i>  | Невозможно | Возможно   | Возможны   |
| <i>REPEATABLE READ</i> | Невозможно | Невозможно | Возможны   |
| <i>SERIALIZABLE</i>    | Невозможно | Невозможно | Невозможны |

### Завершение транзакций

Как мы отмечали в начале этого раздела, транзакции могут инициироваться как явным способом (с помощью оператора *START TRANSACTION*), так и неявно, при выполнении первого оператора, требующего наличия контекста транзакции. Для завершения транзакции всегда<sup>16</sup> требуется выполнение одного из двух операторов *COMMIT* (фиксация транзакции) или *ROLLBACK* (откат транзакции), которые имеют следующий синтаксис:

```
COMMIT [WORK] [AND [NO] CHAIN]
ROLLBACK [WORK] [AND [NO] CHAIN]
[TO SAVEPOINT savepoint_name]
```

При желании завершить транзакцию таким образом, чтобы все произведенные ею изменения были навсегда сохранены в базе данных, следует завершать транзакцию оператором *COMMIT* (как видно из синтаксиса, допускается эквивалентный вид *COMMIT WORK*). Если требуется завершить транзакцию с *аннулированием* всех произведенных изменений, то нужно использовать оператор *ROLLBACK ( ROLLBACK WORK )*.

Заметим, что и операция *фиксации транзакции*, и операция отката являются достаточно сложными и выполняются не мгновенно. Поэтому в ходе выполнения этих операций, вообще говоря, может произойти аварийный отказ системы. Естественно (хотя в этом курсе мы не обсуждаем технические детали возможных реализаций), база данных будет восстановлена в свое последнее *согласованное* состояние, но ситуации прерванного выполнения операции *фиксации* и операции отката коренным образом различаются. Оператор *COMMIT* считается безусловно выполненным только в том случае, когда *сервер баз данных* подтвердил это после выполнения всех действий, требуемых для *фиксации транзакции*. Аварийная ситуация во время выполнения операции *ROLLBACK* ничем не отличается от аварийной ситуации, возникшей в процессе выполнения транзакции. В этом случае (при восстановлении базы данных) прерванная транзакция считается незафиксированной (что так и есть), и все ее изменения автоматически удаляются из состояния базы данных. Поэтому окончательный результат выполнения операции *фиксации транзакции*, прерванной аварийным отказом системы, эквивалентен успешному выполнению операции отката транзакции<sup>17</sup>.

Синтаксис обоих операторов показывает, что в каждом из них может содержаться раздел *AND [ NO ] CHAIN*. Постараемся кратко пояснить смысл *этого раздела* (не вдаваясь в детали, поскольку *стандарт SQL:1999* оставляет окончательное решение за реализацией).

Операции образования и завершения транзакции являются достаточно дорогостоящими. В особенности это касается операции завершения транзакции, при выполнении которой необходимо выполнить обмены с внешней памятью. С другой стороны, использование

долговременных транзакций чревато снижением уровня *параллелизма* в системе, а также повышает риск утраты результатов транзакции в результате системного отказа.

Поэтому часто используется компромиссный вариант, при котором действия операторов **COMMIT** или **ROLLBACK** приводят не только к завершению текущей транзакции, но и к образованию новой транзакции<sup>18</sup>. Именно эту возможность поддерживает раздел **AND [ NO ] CHAIN** операторов **COMMIT** и **ROLLBACK**. Если такой раздел отсутствует в операторе завершения транзакции, то подразумевается наличие раздела **AND NO CHAIN**, и новая транзакция не образуется. Если же раздел **AND CHAIN** присутствует, то немедленно после завершения выполнения **COMMIT** или **ROLLBACK** текущей транзакции образуется новая транзакция, наследующая все характеристики завершенной транзакции.

Семантику раздела **TO SAVEPOINT** мы поясним немного позже.

### Транзакции и ограничения целостности

Материал этого подраздела уже излагался в лекции 13, но там это делалось в контексте определений ограничений целостности. Для полноты картины мы воспроизведем часть этого материала в контексте *управления транзакциями*.

Итак, любое *ограничение целостности* обладает атрибутом, определяющим время проверки данного ограничения. Этот атрибут может иметь значения **DEFERRABLE** (отложенная проверка) или **NOT DEFERRABLE** (немедленная проверка). Чтобы данное *ограничение целостности* могло когда-либо обладать свойством отложенной проверки, нужно, чтобы в определении такого ограничения присутствовали ключевые слова **INITIALLY DEFERRED** или **INITIALLY IMMEDIATE**. В любом случае, в каждый момент времени выполнения транзакции любое *ограничение целостности* находится в одном из двух состояний - **отложенная проверка** или **немедленная проверка**. Если начальным состоянием ограничения является **INITIALLY DEFERRED**, то в начале любой транзакции его текущим состоянием будет **отложенная проверка**. Аналогично для ограничений с начальным состоянием **INITIALLY IMMEDIATE**.

Любое ограничение, находящееся в состоянии **немедленной проверки**, всегда проверяется в конце выполнения любого оператора SQL<sup>19</sup>. Немедленно проверяются и те ограничения, которые были определены как **NOT DEFERRABLE**, но для которых впоследствии был установлен режим немедленной проверки. Однако если текущим состоянием ограничения является **отложенная проверка**, оно будет проверяться только тогда, когда перейдет в состояние **немедленной проверки**. Это делается неявно при выполнении оператора **COMMIT** или явно при выполнении оператора **SET CONSTRAINTS**. Этот оператор имеет следующий синтаксис:

```
SET CONSTRAINTS { ALL | constraint_name_commalist }
{ DEFERRED | IMMEDIATE }
```

Ключевое слово **ALL** является сокращенной формой задания списка имен всех ограничений целостности, определенных в базе данных, которые специфицированы с указанием ключевого слова **DEFERRABLE**. Если список имен ограничений задается явно, то все входящие в него имена должны соответствовать ограничениям, определенным с указанием ключевого слова **DEFERRABLE**.

При попытке *фиксации транзакции*, для которой имеются одно или несколько ограничений целостности, текущим режимом которых является **отложенная проверка**, система (ненадолго, поскольку транзакция скоро тем или иным способом завершится) устанавливает для всех этих ограничений режим немедленной проверки и проверяет ограничения. Если какое-либо из ограничений нарушается, то операция **COMMIT** трактуется как операция **ROLLBACK**, и пользователю (или приложению) сообщается, что возникла ошибка. Избежать этой неприятной ситуации можно явным выполнением оператора **SET CONSTRAINTS ALL IMMEDIATE** до *фиксации транзакции*, для которой имеются **DEFERRABLE** ограничения, текущим режимом которых является отложенная проверка.

## Точки сохранения

Как мы уже отмечали, использование долговременных транзакций повышает риск полного *аннулирования* результатов транзакции по причине нарушения ограничений с отложенной проверкой при выполнении каких-либо экспериментальных (недостаточно проверенных) операций. Конечно, теоретически можно было бы оформлять выполнение каждой такой подозрительной операции в виде отдельной транзакции, но это часто противоречит общей логике приложения, когда последовательность действий должна быть *атомарной*.

Частичное решение этой проблемы предоставляет механизм **точек сохранения (savepoint)** SQL:1999. Точка сохранения представляет собой своего рода пометку в последовательности операций транзакции, которую в дальнейшем можно использовать для частичного отката транзакции с сохранением жизнеспособности транзакции и результатов операций, выполненных в транзакции до точки сохранения. Пример использования точки сохранения показан на [рис.18.7](#).



**Рис. 18.7.** Пример транзакции с точкой сохранения

На этом рисунке после выполнения последовательности проверенных "безопасных" операций, которые, по мнению пользователя, не могут нарушить *ограничения целостности* с отложенной проверкой, устанавливается точка сохранения. За этой точкой следует серия "рискованных" операций. Если по каким-то причинам (например, путем немедленной проверки отложенных ограничений) затем принимается решение о нецелесообразности фиксации результатов данных операций, то выполняется частичный откат транзакции к точке сохранения, а затем фиксируются результаты безопасных операций.

Допускается установка в одной транзакции нескольких последовательных точек сохранения. При установке каждой точки сохранения ей назначается некоторое (локальное в пределах транзакции) имя, которое в дальнейшем может использоваться в операции **ROLLBACK** для задания точки частичного отката транзакции (см. выше синтаксис *оператора ROLLBACK*). Если последовательно устанавливаются две точки сохранения **SP1** и **SP2** и затем выполняется операция **ROLLBACK TO SAVEPOINT SP1**, то восстановление производится до **SP1** (через **SP2**), и точка сохранения **SP2** "забывается"<sup>20</sup>.

Для установления точки сохранения используется оператор **SAVEPOINT** с очевидным синтаксисом

**SAVEPOINT** *savepoint\_name*

Можно также отказаться от ранее установленной точки сохранения, удалив ее из контекста транзакции. Для этого предназначен оператор **RELEASE**, синтаксис которого также очевиден:

**RELEASE SAVEPOINT** *savepoint\_name*

После выполнения этой *операции в данной* транзакции невозможно выполнять какие-либо другие операции над точкой сохранения с данным именем, пока не будет образована другая одноименная точка сохранения.

### Подключения и сессии

Может показаться странным, что мы оставили на конец этой лекции материал, который, казалось бы, необходимо знать, чтобы иметь возможность приступить к работе с какой-либо из современных систем баз данных. Объяснение очень простое. Чем ниже уровень средств языка **SQL**, чем ближе эти средства соприкасаются с индивидуальными особенностями реализаций, тем менее точен и конкретен стандарт **SQL**. А в данном разделе речь идет о средствах, реализация которых в **СУБД** разных поставщиков обладает очень большой спецификой.

Сильно упрощая текущую ситуацию, можно сказать, что практически все современные продукты управления **SQL**-ориентированными базами данных основаны на архитектуре "клиент-сервер". Принципиальная *схема* клиент-серверной организации показана на [рис.18.8](#).

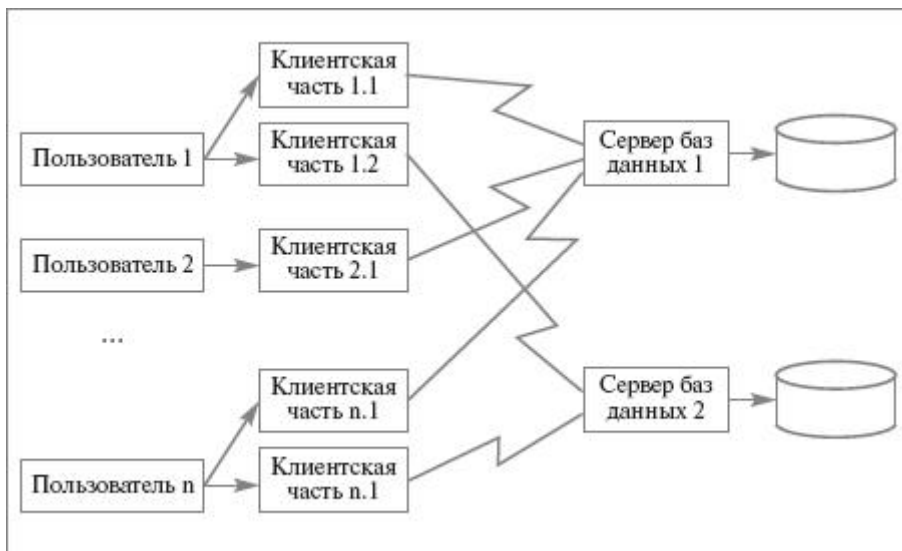


**Рис. 18.8.** Клиент-серверная архитектура СУБД

Конечно, это рисунок весьма условен. Под термином **пользователь** здесь, конечно, понимается некоторое *приложение*, с которым реально работает *конечный пользователь* (например, в этом приложении может быть реализован *монитор* прямого SQL). **Клиентская часть СУБД** - это тот системный *компонент*, с которым непосредственно взаимодействует **пользователь**. Данный *компонент* скрывает специфику реальных взаимодействий с серверной частью СУБД (например, используемые *сетевые протоколы*, если клиентская и серверная части СУБД разнесены по разным компьютерам сети). Наконец, **сервер баз данных** представляет собой основную часть СУБД, где, собственно, и происходит выполнение операторов SQL и осуществляется *доступ* к базе данных.

Важно обратить внимание, что *программные компоненты*, представляющие **пользователя** и **клиентскую часть СУБД**, обычно выполняются на одном компьютере, а **сервер баз данных** работает на другом (серверном) компьютере. Но вполне может быть, что все три перечисленных программных компонента в действительности размещены на одном компьютере.

### Установление соединений

Начиная со стандарта SQL/92, при разработке языковых средств стала приниматься во внимание клиент-серверная организация СУБД. Если говорить более точно, стал очевиден тот факт, что во всех существующих СУБД до начала работы приложения со средствами управления базой данных требуется выполнить некоторые предварительные иницилирующие действия. В частности, необходимо создать контекст, в котором будет работать система баз данных. В некоторых реализациях этот контекст создается автоматически при запуске приложения, поскольку клиентская часть СУБД компонуется к приложению. В других случаях прикладная программа связывается с СУБД за счет наличия специализированных реализационно зависимых средств подключения к СУБД. Иногда контекст формируется на основе состояния *системных переменных*.

Очевидно, что для выработки языковых средств, которые не противоречили бы существующим реализациям, требовался компромисс. Этот компромисс выразился в том, что в SQL:1999 допускается установление связи приложения с СУБД по умолчанию, а

также обеспечиваются средства явного управления соединениями. Общий подход состоит в следующем.

- Почти все операторы SQL (с небольшим числом исключений) могут выполняться только при наличии подключения клиентской части СУБД к серверу базы данных.
- Если соединение с сервером *установлено* и приложение пытается выполнить один из операторов SQL (для выполнения которых требуется соединение), то его выполняет та СУБД, с которой *установлено соединение*.
- Если приложение пытается выполнить один из операторов SQL (для выполнения которых требуется соединение), а соединение не *установлено*, то, прежде всего, требуется *установить соединение*. В SQL:1999 указывается, что такое соединение является соединением с СУБД по умолчанию. Что собой представляет это умолчание, определяется в реализации. После *установления соединения* упомянутый оператор SQL выполняется той СУБД, с которой *установлено соединение*.
- Если первым (до *установки соединения*) выполняемым оператором SQL является оператор **CONNECT** (это одно из исключений), то соединение по умолчанию не устанавливается, а происходит обращение к запрашиваемому серверу, и соединение устанавливается именно с ним.
- Можно выполнять оператор **CONNECT** для *установления соединений* со вторым, третьим и т. д. серверами, не разрывая ранее *установленные соединения*. Каждое вновь *установленное соединение* называется **текущим соединением (current connection)**, а все ранее *установленные соединения* - **отложенными соединениями (dormant connection)**.
- С каждым соединением ассоциирована сессия. Сессия, ассоциированная с текущим соединением, называется **текущей сессией (current session)**, а сессии, ассоциированные с отложенными соединениями, называются **отложенными сессиями (dormant session)**<sup>21</sup>
- Если у приложения имеется несколько соединений, можно переключать их с помощью оператора **SET CONNECTION**.
- Для поддержания *установленных соединений* могут расходоваться значительные *системные ресурсы*. Поэтому может возникнуть потребность в ликвидации соединения. Это можно сделать с помощью оператора **DISCONNECT**. Все соединения, не ликвидированные явно до завершения работы приложения, ликвидируются системой автоматически. Попытка ликвидировать текущее соединение, в котором выполняется транзакция, расценивается как ошибка.
- В реализации определяется, можно ли переключать соединения во время выполнения транзакции. Однако если реализация это допускает, то, в соответствии со стандартом, все операторы, выполняемые в одной транзакции, но в разных соединениях, являются частью одной общей транзакции.

### Операторы SQL для управления соединениями

Как отмечалось выше, в эту группу входят операторы **CONNECT**, **SET CONNECTION** и **DISCONNECT**.

#### Оператор CONNECT

Оператор определяется следующими синтаксическими правилами:

```
CONNECT TO connection_target
connection_target ::= SQL_server_name
```

[ AS connection\_name ]  
[ USER connection\_user\_name ]  
| DEFAULT

Здесь **SQL\_server\_name** - это литерально заданная *символьная строка*, идентифицирующая сервер, к которому требуется подключиться. Смысл (и формат) этого имени определяется в реализации.

В необязательном разделе **AS** указываемое имя (**connection\_name**) выступает в роли временного имени соединения, которое впоследствии может быть использовано в операторах **SET CONNECTION** и **DISCONNECT**. Если в операторе **CONNECT** раздел **AS** не содержится, то по умолчанию **connection\_name** совпадает с **SQL\_server\_name**.

В необязательном разделе **USER** указываемое имя (**connection\_user\_name**) идентифицирует пользователя, от имени которого устанавливается соединение. При отсутствии раздела **USER** в качестве **connection\_user\_name** по умолчанию принимается текущий **authID**. В стандарте допускается, что реализация может ограничить возможные значения **connection\_user\_name** (например, потребовать, чтобы это имя всегда совпадало с текущим **authID**).

Эффект использования оператора в форме **CONNECT TO DEFAULT** почти не отличается от результата действия системы при отсутствии какого-либо явного требования соединения. (Напомним, что соединение по умолчанию неявно устанавливается при попытке выполнения первого оператора SQL, требующего соединения.) Однако имеется одно важное отличие. Если соединение по умолчанию устанавливается неявно, а затем вдруг прерывается из-за какой-то ошибки, то оно автоматически переустанавливается при выполнении следующего оператора SQL. Если же соединение по умолчанию устанавливается явным образом, то автоматическое повторное *установление соединения* после его разрыва не производится.

## Оператор SET CONNECTION

Оператор определяется следующими синтаксическими правилами:

**SET CONNECTION connection\_object**  
**connection\_object ::= { connection\_name | DEFAULT }**

Условием успешного выполнения операции является наличие отложенного *установленного соединения* с именем **connection\_name** или отложенного *установленного соединения* по умолчанию. В этом случае текущее соединение становится отложенным, а указанное отложенное соединение - текущим.

## Оператор DISCONNECT

Оператор имеет следующий синтаксис:

**DISCONNECT { connection\_object | ALL | CURRENT }**

Необходимым условием для возможности ликвидации соединения является отсутствие *активной транзакции* в этом соединении.

Если в операторе указывается **connection\_object**, то соответствующее имя должно соответствовать *установленному* (текущему или отложенному) соединению. Если указывается **CURRENT**, то должно существовать текущее соединение.

Если оператор применяется к текущему соединению, то это соединение ликвидируется, и ни одно соединение не является текущим. В таком случае для продолжения работы необходимо *установить текущее соединение* при помощи операторов **CONNECT** или **SET CONNECTION**.

Если в операторе указывается **ALL**, то ликвидируются все соединения, включая текущее.

## Заключение

В этой лекции были рассмотрены три темы, которые являются относительно независимыми, но относятся к средствам языка **SQL**, предназначенным для регулирования доступа пользователей к базам данных. На первый взгляд материал этой лекции проще материала предыдущих лекций, посвященных языку **SQL**. Наверное, это действительно так, если говорить про чисто языковую сложность соответствующих операторов **SQL**. Но в действительности (которую мы старательно обходили в основных разделах лекции) дело обстоит гораздо сложнее.

Как легко видеть, при распространении *привилегий* и *ролей* могут возникать произвольно сложные ориентированные графы связей между объектами *базы данных*, владельцами *привилегий*, *привилегиями* и *ролями*. Если изображать сплошными стрелками *передачу привилегий*, прерывистыми - *передачу ролей*, пунктирными - владение *привилегиями*, а точечными - владение *ролями*, то даже по отношению к одной *привилегии pr* для одного объекта *o* может появиться следующий *граф связей* (**userID** означает **authID**, отличный от имени *роли*), показанному на [рис.18.8](#).

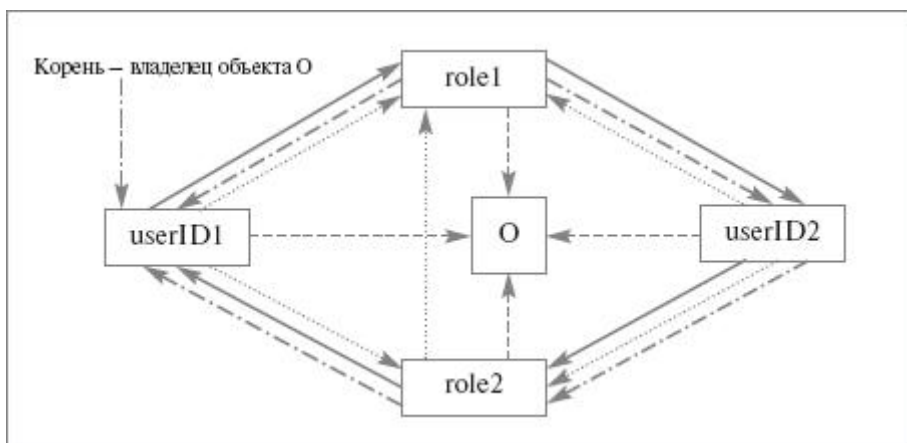


**Рис. 18.9.** Простейший граф идентификаторов пользователя, имен ролей, объектов и привилегий

Как мог появиться такой *граф*? Пользователь с **authID**, равным **userID1** (это мы предположили для упрощения, а вообще-то это могло быть и именем *роли*), создает объект **o**, становится его владельцем и тем самым обладателем *привилегии pr* по отношению к этому объекту. Пользователь **userID1** предоставляет полномочие **pr** роли **role1** (с правом *передачи*). Затем пользователю **userID1** предоставляется роль **role1** (с правом *передачи*), и он получает право исполнять эту роль. От имени роли **role1** полномочие **pr** передается

пользователю **userID2** (с правом *передачи*), и этот же *пользователь* получает право исполнять *роль* **role1** (с правом *передачи*). Пользователь **userID2** передает *роли* **role2** *роль* **role1** и полномочие **pr** (с правом *передачи*). Наконец, от имени *роли* **role2** полномочие **pr** и сама *роль* **role2** передаются пользователю **userID1**.

Попробуйте теперь проследить, как будет выполняться операция

**REVOKE pr ON o FROM role1 CASCADED**

Какие узлы и дуги останутся в графе? Задача не очень сложная, но, очевидно, нетривиальная. И такого рода задачи приходится ежедневно решать администраторам больших и динамических SQL-ориентированных баз данных.

Теперь немного поговорим об управлении транзакциями. В *стандарте SQL:1999* ничего не говорится о возможной реализации различных *уровней изоляции*. Конечно, это правильно, поскольку спецификация языка не должна накладывать какие-либо ограничения на реализации. Но, к сожалению, при использовании SQL-ориентированной *СУБД* некоторые знания о реализации механизма транзакций необходимы. Например, предположим, что имеются две транзакции **T1** и **T2**, выполняемые в режиме изоляции **SERIALIZABLE**. Предположим, что они должны работать по "симметричному" плану, показанному на [рис.18.9](#).

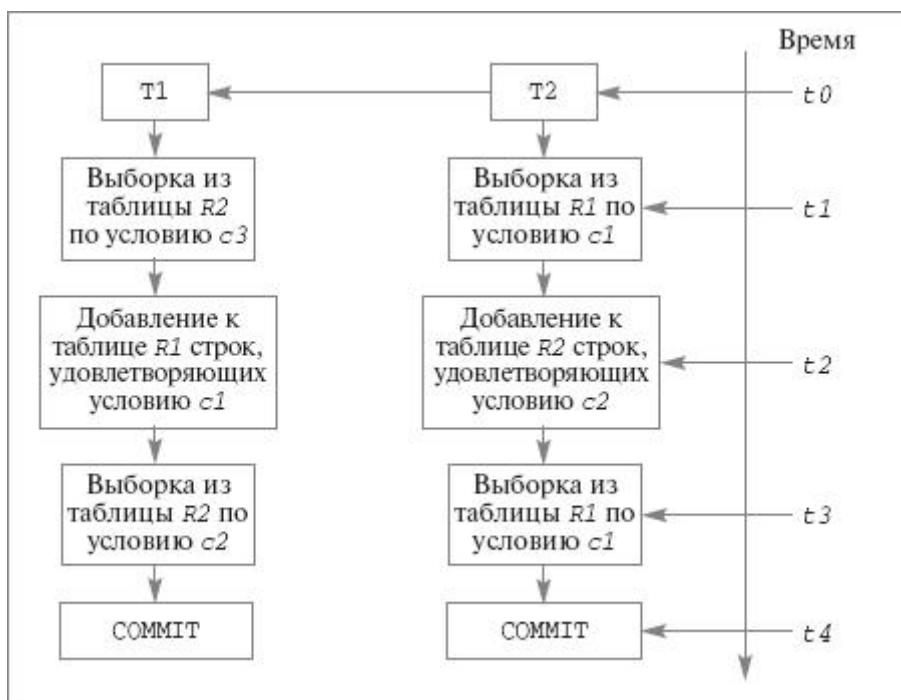


**Рис. 18.10.** Взаимные "фантомы"

Транзакции работают в наивысшем режиме изолированности. Эффект их выполнения должен быть эквивалентен эффекту некоторого последовательного выполнения транзакций **T1** и **T2**. Но попробуйте придумать какой-либо корректный способ одновременного выполнения этих транзакций, который привел бы к эффекту их последовательного выполнения. Другими словами, для грамотного использования механизма транзакций на уровне языка SQL необходимо знать, каким образом данный механизм реализован в используемой *СУБД*.

И, конечно же, знания о реализации абсолютно необходимы при использовании механизма подключений и сессий. Слишком много в этой части стандарта отдается на волю реализации.

## Лабораторная работа9. Объектные расширения

### Введение

Как отмечалось в "[Общее введение, типы данных и средства определения доменов](#)", язык *SQL* появился в середине 1970-х гг. при выполнении экспериментального проекта реляционной СУБД *System R*. Проект выполнялся в компании *IBM*, и это вполне естественно, потому что именно сотрудник *IBM* Эдгар Кодд предложил миру идею реляционных баз данных. От *System R* исходит большинство традиционных средств стандарта *SQL:1999* (и *SQL:2003*), которые мы обсуждали в восьми предыдущих лекциях. Однако в этой лекции речь пойдет о возможностях современных вариантов *SQL*, которые не имеют отношения к *System R* (за исключением некоторых экспериментов по представлению сложных объектов средствами *SQL*) и, вообще говоря, к реляционной модели данных, а именно — о так называемых объектно-реляционных расширениях языка. Чтобы у читателей не возникло впечатление, что объектные расширения появились в языке *SQL* внезапно, благодаря какому-то озарению разработчиков языка, мы начнем эту лекцию с небольшого (и крайне субъективного) очерка истории объектно-реляционного подхода к организации систем баз данных.

### Истоки и краткая история объектно-реляционных баз данных

Пальму первенства в области объектно-реляционных систем управления базами данных (ОРСУБД) оспаривают два весьма известных специалиста в области технологии баз данных – Майкл Стоунбрейкер (Michael Stonebraker) и Вон Ким (Won Kim).

### Первые ОРСУБД

Майкл Стоунбрейкер начал работать в области баз данных в начале 1970-х гг. прошлого века в университете Беркли. Его первым всемирно известным проектом была реляционная СУБД *Ingres*, которая существует и используется до сих пор в двух ипостасях – как свободно распространяемая система (университетская *Ingres*; код поддерживается в Беркли) и как коммерческая СУБД, принадлежащая компании *Computer Associates*. В исходном варианте СУБД *Ingres* отсутствовала поддержка языка *SQL* (поддерживался собственный язык запросов *QUEL*), но система уже обладала некоторыми уникальными чертами, которые, с небольшой натяжкой, можно было бы назвать объектными (например, в СУБД *Ingres* допускалось определение пользовательских процедур, выполняемых на стороне сервера). Кроме того, в проекте *Ingres* очень большое внимание уделялось управлению правилами.

В 1980-е гг. Майкл Стоунбрейкер возглавлял проект *Postgres* (вариант этой системы под названием *PostgreSQL* в настоящее время является весьма популярным свободно доступным продуктом). В *Postgres* были реализованы многие интересные средства: поддерживалась темпоральная модель хранения и доступа к данным, и в связи с этим был полностью пересмотрен механизм журнализации изменений, откатов транзакций и восстановления БД после сбоев; обеспечивался мощный механизм ограничений целостности; поддерживались ненормализованные отношения (работа в этом направлении началась еще в среде *Ingres*), хотя и довольно странным способом: в поле отношения мог храниться динамически выполняемый запрос к БД.

Одно свойство системы *Postgres* сближало ее со свойствами объектно-ориентированных СУБД (ООСУБД). В *Postgres* допускалось хранение в полях отношений данных абстрактных, определяемых пользователями типов. Это обеспечивало возможность

внедрения поведенческого аспекта в БД, т. е. решало ту же задачу, что и ООСУБД, хотя, конечно, семантические возможности *модели данных* Postgres были существенно слабее, чем *объектно-ориентированных моделей данных*. Основная разница состояла в том, что в Postgres не предполагалось наличие языка программирования, одинаково понимаемого как *внешней системой программирования*, так и *системой управления базами данных*. Как и в Ingres, в исходном варианте Postgres не поддерживался язык SQL (имелся собственный *язык запросов* Postquel). Кстати, во времена Postgres Майкл Стоунбрейкер не использовал термин *объектно-реляционная система*, предпочитая называть свою СУБД *системой следующего поколения*.

В начале 1990-х гг. Стоунбрейкер создал *компанию Illustra*, основной целью которой был выпуск коммерческого варианта *СУБД Postgres*, получившего название *Illustra*. В этой системе поддерживались основные идеи Postgres, но уже присутствовала и поддержка языка SQL. В конце 1995 г. *компания Illustra* была поглощена *компанией Informix*, и это привело к выпуску в 1996 г. *СУБД Informix Universal Server* (см. ниже).

Имя Вона Кима стало широко известно во второй половине 1970-х гг., когда он примкнул к участию в экспериментальном проекте *компании IBM System R*. Наиболее известная ранняя работа доктора Кима была посвящена преобразованию *SQL-запросов* с целью превращения запросов с вложенными *подзапросами* в запросы с соединениями.

В 1980-е гг. Вон Ким работал в *компании MCC*, где успешно выполнил реализацию серии прототипов *ООСУБД Orion*. В этих прототипах были опробованы многие идеи *объектно-ориентированных СУБД*. Одной из интересных особенностей проекта было то, что в качестве основного языка программирования использовался *объектный вариант* известного функционального языка *Lisp*.

В конце 80-х гг. д-р Ким основал *компанию UniSQL*, выпустившую в 1991 г. первую версию продукта *UniSQL*, который Вон Ким стал называть *объектно-реляционной системой*. Трудно оценивать коммерческий успех этой СУБД. В настоящее время она принадлежит Корейской национальной телекоммуникационной компании и, по всей видимости, продолжает использоваться. Поскольку *UniSQL* была первой СУБД, официально называемой *объектно-реляционной системой*, приведем ее краткое описание.

*UniSQL* обеспечивала возможность построения так называемых *федеративных систем баз данных*. При этом обеспечивалось единое представление данных, которые могли храниться либо в базе данных, непосредственно управляемой *UniSQL*, либо в какой-либо из *реляционных баз данных*, управляемой СУБД *Oracle, Informix, Sybase* и т. д., либо в какой-либо *дореляционной базе данных*. Сервер *UniSQL* обеспечивал интегрированный доступ к данным, управляемым разными СУБД. Одна из возможных конфигураций использования системы показана на [рис. 19.1](#).

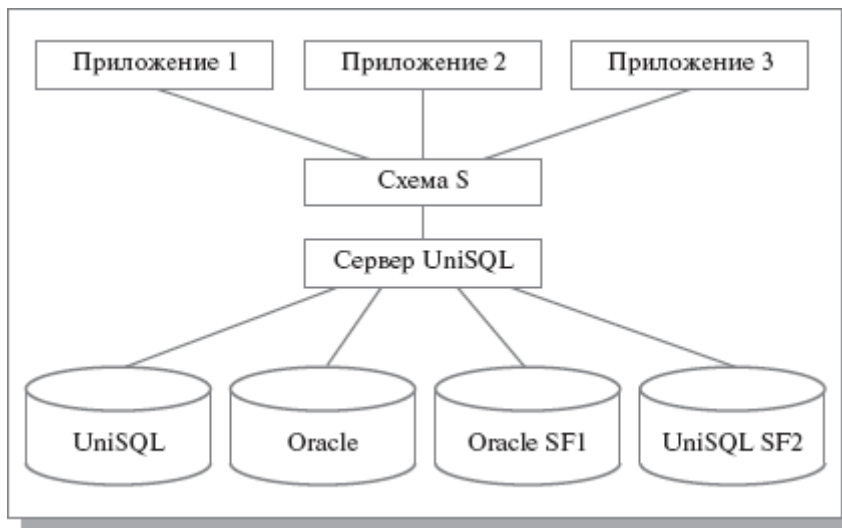


**Рис. 19.1.** Возможная конфигурация системы UniSQL

Как показывает [рис. 19.1](#), сервер *UniSQL* позволяет *представлениям* работать через "глобальную" *схему базы данных S*, полученную из двух "фрагментарных" *схем баз данных*, которые управляются непосредственно *UniSQL* и СУБД Oracle<sup>1</sup>.

Разработчики *UniSQL* полагали, что построение полнофункциональной СУБД, основанной на принципиально новой *модели данных*, крайне проблематично. Был выбран подход к расширению реляционной модели, выражающийся в следующих четырех принципах:

- значения атрибутов отношений могут быть не только литеральные значения, но и объекты;
- значения атрибутов отношений не обязательно являются *атомарными*;
- при построении таблиц (классов) может использоваться механизм наследования;
- классы включают операции.

В созданной компанией системе поддерживалось расширение стандарта SQL – SQL/X, одновременно включающее и объектно-ориентированные, и реляционные возможности. В одном языке поддерживались возможности и определения данных, и манипулирования ими. В качестве языковых средств *программирования приложений* поддерживались языки C++ и *Smalltalk*.

### **Внедрение объектных расширений в основные РСУБД**

В конце 1989 г. группа известных специалистов в области языков программирования баз данных опубликовала документ под названием *Манифест систем объектно-ориентированных баз данных* (для краткости будем называть его *Первым манифестом*). Основным поводом к написанию и публикации этого материала было то, что к тому времени существовал ряд систем управления базами данных, которые, по большому счету, объединяла только общая привязанность к объектно-ориентированным языкам программирования. В *Первом манифесте* была предпринята попытка дать *определение системам объектно-ориентированных баз данных*. Авторы стремились привести описание основных свойств и характеристик, которыми должна обладать система, претендующая на то, чтобы называться системой *объектно-ориентированных баз данных*. *Первый манифест* был написан академическими исследователями; почти все они

являлись и являются профессорами различных университетов. Конечно, это нашло свое отражение в стиле *Первого манифеста* – очень мягком и умеренно рекомендательном (хотя по своему духу предложения этого *манифеста* были весьма радикальными).

Через год после публикации *Первого манифеста* вышел в свет *Манифест систем баз данных третьего поколения*, инициатором которого, очевидно, был Майкл Стоунбрейкер (хотя у документа формально имелось много авторов). Мы говорим об этом с уверенностью, поскольку в этом *манифесте* повсюду видны идеи Стоунбрейкера, использованные им в проектах Ingres и Postgres.

В некотором роде *Манифест систем баз данных следующего поколения* (для краткости мы будем называть его *Вторым манифестом*) стал ответом миру *объектно-ориентированных баз данных* со стороны мира *SQL-ориентированных баз данных*. Если *Первый манифест* был хотя и немного путанным, но все-таки носил научный характер, то *Второй манифест* является в большей степени инженерно-публицистическим документом. *Второй манифест* можно расценивать как реакцию индустрии СУБД на неприятные для нее измышления науки.

*Второй манифест* (или, вернее, работы, приведшие к его появлению) имел важные последствия. В 1995 г. компания Informix (ныне входящая в состав IBM) купила компанию Майкла Стоунбрейкера Illustra<sup>2</sup>, и Стоунбрейкер стал техническим директором Informix. В начале 1996 г. компания Informix объявила о выпуске принципиально нового продукта Informix Universal Server, в котором, как утверждалось, лучшие черты Informix Online Server сочетались с развитыми объектными чертами, присущими Illustra.

К выпуску Informix Universal Server очень ревниво отнеслась компания Oracle, которая немедленно заявила, что у нее готов собственный объектно-реляционный продукт, по всем параметрам превосходящий систему компании Informix. Эта система, получившая название Oracle8, была выпущена в конце лета 1996 г.

Годом позже к группе производителей объектно-реляционных СУБД (ОРСУБД) примкнула компания IBM, выпустившая продукт DB2 Universal Database. Как выяснилось позже, все наиболее важные свойства этого продукта были реализованы еще в 1995 г. в СУБД DB2 for Common Servers. Просто компания IBM предпочла до поры не афишировать свои расширения.

Первые пару лет вокруг объектно-реляционных СУБД стоял большой шум. Позже выяснилось, что маркетинговые ожидания компаний гигантов оказались преувеличенными. (В частности, это было одной из основных причин падения компании Informix.) Сегодня объектные расширения SQL-ориентированных СУБД предлагаются пользователям лишь в качестве дополнительных, хотя и важных возможностей.

Объектные расширения языка SQL были зафиксированы в стандарте SQL:1999. В той или иной мере эти расширения поддерживаются во всех трех перечисленных выше продуктах. В настоящее время ближе всех к стандарту находятся СУБД компании Oracle и DB2 компании IBM.

## **Объектная модель SQL**

Объектные расширения SQL:1999 базируются на некоторой объектной модели, хотя эта модель в явном виде в стандарте не специфицируется. Объектная модель SQL<sup>3</sup> не является

тождественной объектным моделям какого-либо объектно-ориентированного языка программирования или какой-либо объектно-ориентированной системы баз данных. Однако при определении объектной модели SQL участники процесса стандартизации тщательно проанализировали ряд других языков и систем с целью выяснения достоинств и недостатков их *объектных моделей*. По мнению авторов *стандарта SQL:1999*, выработанная ими *объектная модель* похожа на *объектную модель* языка Java<sup>4</sup>, но при этом адаптирована к природе языка SQL как языка СУБД с наличием стабильно хранимых *метаданных* и данных.

*Объектная модель SQL:1999* включает два основных отличительных компонента – *структурные*, определяемые пользователями типы данных (User Defined Type – UDT) и *типизированные таблицы* (Typed Table). Первый компонент позволяет определять новые типы данных, которые могут быть гораздо более сложными, чем *встроенные типы данных* языка SQL. При определении *структурного UDT* требуется специфицировать не только содержащиеся в нем элементы данных, но и семантику типа данных, т. е. его поведение на основе интерфейса вызовов методов. Второй компонент – *типизированные таблицы* – позволяет определять таблицы, строки которых являются экземплярами (или значениями) UDT, с которым явно ассоциируется таблица. Во многих отношениях строка *типизированной таблицы* похожа на объект класса в объектно-ориентированной системе.

В *стандарте SQL:1999* определены два пакета объектных свойств – *минимальный* (PKG006) и *полный* (PKG007), которым должны удовлетворять SQL-ориентированные ОПСУБД, претендующие на соответствие стандарту. Ниже будут перечислены свойства, включенные в каждый из пакетов, но смысл этих свойств будет понятен только после прочтения остальных разделов.

*Пакет PKG006* включает всего пять свойств:

- свойство S023 ("Basic structured types") – возможность определять UDT и их методы с ограниченными возможностями;
- свойство S041 ("Basic reference types") – возможность определять и использовать ссылки на экземпляры UDT, входящие в *типизированные таблицы* ;
- свойство S051 ("Create table of type") – возможность создания *типизированных таблиц* ;
- свойство S151 ("Type predicate") – возможность определения точного типа (в *иерархии типов*) экземпляра UDT;
- свойство T041 ("Basic LOB data type support") – возможность определения *LOB-типов* в смысле SQL (с необязательной поддержкой операций, кроме операций сохранения и полной выборки).

*Пакет PKG007* содержит девять дополнительных свойств:

- свойство S024 ("Enhanced structured types") добавляет к свойству S023 ряд развитых возможностей, в число которых входят возможности кодирования методов на языках, отличных от SQL, сравнения экземпляров UDT и передача экземпляров UDT в качестве параметров различных процедур;
- свойство S043 ("Enhanced reference types") расширяет свойство S041 возможностями определения ссылок с областью действия, автоматической проверки законности ссылок и т. д.;
- свойство S071 ("SQL-paths in function and type name resolution") позволяет использовать *путевые выражения SQL* (SQL-path) в алгоритме разрешения типа;

- свойство S081 ("Subtables") расширяет возможности свойства S051, допуская организацию иерархии таблиц, аналогичной *иерархии типов* соответствующих UDT;
- свойство S111 ("ONLY in query expressions") обеспечивает возможность выборки только экземпляров указанного типа, без экземпляров любого из его *подтипов*;
- свойство S161 ("Subtype treatment") позволяет информировать среду SQL о том, что некоторый экземпляр UDT в действительности является экземпляром указанного *подтипа*;
- свойство S211 ("User-defined cast functions") разрешает определять подпрограммы, преобразующие экземпляры UDT к другим типам;
- свойство S231 ("Structured type locators") способствует доступу к экземплярам UDT из прикладных программ;
- свойство S023 ("Transform functions") позволяет определять подпрограммы, преобразующие значения UDT в значения предопределенных типов данных, и наоборот.

### Цели лекции

Следует заметить, что в этой, безусловно, перегруженной материалом лекции мы преследуем две основные цели. Первая цель состоит в том, чтобы показать читателям, что в средствах определения *структурных типов* SQL используются, по сути, все базовые возможности определения *объектов базы данных* и выборки данных, которые обсуждались в предыдущих лекциях. Более того, определенные пользователями типы в SQL являются объектами первого класса; UDT можно использовать в любой конструкции языка, в которой допускается применение предопределенного или конструируемого типа данных. Очень важно отдавать себе отчет в том, что наличие возможности определять *пользовательские типы* не делает язык SQL менее реляционным или более объектным. Эта возможность "всего лишь" фантастически повышает выразительную мощность языка.

Второй целью является демонстрация того, как на основе базовых механизмов языка удалось ввести дополнительные конструкции, которые действительно вплотную приближают SQL к объектному миру. Здесь, конечно, основную роль играет связка UDT и механизма *типизированных таблиц*, которые играют в SQL своеобразную *совмещенную роль* классов и *коллекций объектов*.

Может оказаться, что материал этой лекции покажется сложным, поскольку для его усвоения не помешало бы иметь предварительную подготовку в области полнотиповых языков программирования, объектно-ориентированных языков и систем баз данных и т. д. Хочется надеяться, что возникновение трудностей при изучении лекции не отпугнет читателей от этой темы, а напротив, послужит стимулом к изучению дополнительной литературы.

Возможна и другая *опасная ситуация*. Краткость и некоторая формальность изложения может создать ложное впечатление тривиальности объектных расширений SQL. В этом случае могу посоветовать перечитать предыдущие лекции курса, относящиеся к SQL, считая, что везде, где используются предопределенные или конструируемые типы, применяются некоторые UDT, а в тех случаях, где имеются таблицы, связываемые *естественным соединением*, используются *типизированные таблицы*. Думаю, это позволит оценить мощь новых возможностей SQL.

Введя этот необходимый контекст, перейдем к описанию соответствующих механизмов SQL:1999.

## Определяемые пользователями типы

Один из основных упреков по адресу языка SQL, звучавший, в частности, в *Первом манифесте*, заключался в отсутствии каких бы то ни было возможностей хранить в базе данных данные, тип которых являлся бы не предопределенным, а определяемым пользователями. Отрицательные последствия отсутствия такой возможности признавались и во *Втором манифесте*. В SQL:1999 этот дефект был устранен. Как отмечалось в лекции 11, в стандарте поддерживается возможность определения пользователями двух разновидностей UDT – *структурных типов* (structured type) и *индивидуальных типов* (distinct types).

### Индивидуальные типы

Напомним, что *индивидуальным типом* называется UDT, основанный на единственном *встроенном типе* (например, **INTEGER**). Значения такого типа нельзя прямо использовать в операциях соответствующего базового типа, однако допускается явное приведение значений *индивидуального типа* к *базовому типу*. Поясним это на примерах.

Пусть заданы следующие определения *индивидуальных типов*:

```
CREATE TYPE EMP_NO AS INTEGER FINAL;
CREATE TYPE DEPT_NO AS INTEGER FINAL;
CREATE TYPE PRO_NO AS INTEGER FINAL;
```

Таблицу **EMP** можно определить следующим образом (упрощенный вариант):

```
CREATE TABLE EMP (
 EMP_ID EMP_NO,
 EMP_NAME VARCHAR(20),
 DEPT_ID DEPT_NO,
 PRO_ID PRO_NO);
```

Такое определение таблицы приведет к тому, что хотя все три *индивидуальных типа* делены на одном и том же базовом типе **INTEGER**, попытка выполнить запрос

```
SELECT EMP_NAME
FROM EMP
WHERE EMP_ID > DEPT_ID;
```

будет отвергнута системой (и это правильно, поскольку, скорее всего, запрос задан по ошибке). Но если действительно требуется сравнивать идентификаторы служащих с идентификаторами их отделов, то можно воспользоваться конструкцией *явного приведения типа*:

```
SELECT EMP_NAME
FROM EMP
WHERE CAST (EMP_ID TO INTEGER) > CAST (DEPT_ID TO INTEGER);
```

Аналогичным образом будет отвергнут запрос

```
SELECT EMP_NAME, EMP_ID + 5
FROM EMP
WHERE DEPT_ID > 630;
```

Чтобы указать системе, что действительно требуется выполнить операции целочисленного сложения и сравнения над значениями *индивидуальных типов*, запрос нужно переписать следующим образом:

```
SELECT EMP_NAME, CAST (EMP_ID TO INTEGER) + 5
FROM EMP
WHERE CAST (DEPT_ID TO INTEGER) > 630;
```

У читателей могут возникнуть два законных вопроса:

- почему, вопреки обыкновению, мы не привели формальные синтаксические правила операции определения *индивидуального типа*?
- что означает ключевое слово **FINAL** в приведенных примерах определения *индивидуальных типов*?

На оба эти вопроса достаточно дать один (возможно, неожиданный) ответ. С формальной точки зрения *индивидуальный тип* данных является частным случаем *структурного типа* данных. Обе разновидности *UDT* определяются единым синтаксисом, который мы обсудим в следующих подразделах. В частности, ключевое слово **FINAL** играет важную роль в определении *структурного типа*, указывая на тот факт, что этот тип может использоваться только для создания объектов, а не для порождения новых типов на основе механизма наследования. При определении *индивидуальных типов* механизм наследования не используется, и поэтому в определении любого *индивидуального типа* должно присутствовать ключевое слово **FINAL**.<sup>5</sup> Далее, поскольку *индивидуальный тип* является частным типом *структурного типа*, для *индивидуального типа* можно определять методы.

В своих книгах главный редактор стандартов SQL Джим Мелтон постоянно подчеркивает семантическое сходство понятий *индивидуального типа* данных и домена в смысле SQL (Лабораторная работа 11). Более того, утверждается, что в следующих версиях стандарта SQL использование доменов будет сначала объявлено нежелательным, а потом и вовсе будет запрещено. Но я полагаю, что сделать это совсем непросто.

Напомним, что в случае использования SQL-домена:

1. в определении домена указывается базовый *встроенный тип данных* и, возможно, ограничение допустимых значений, которое распространяется на любой столбец, определенный на данном домене;
2. для значений столбца, определенного на домене, допускаются все операции, разрешенные для *базового типа*.

Естественно, эти возможности могут использоваться текущими пользователями стандарта SQL. В то же время в случае использования *индивидуального типа* данных:

1. в определении *индивидуального типа* указывается только базовый тип данных; если столбец определяется на *индивидуальном типе* данных, то для него обязательно придется специфицировать собственное ограничение целостности;

2. для значений столбца, определенного на *индивидуальном типе* данных, не допускаются операции соответствующего *базового типа* (если не использовать операцию явного *приведения типов*).

Здесь явно имеются противоречия, для *сглаживания* которых требуется модифицировать понятие *индивидуального типа* данных.

## Определение структурных типов

Общий синтаксис *оператора определения UDT* ( *индивидуального или структурного* ) определяется следующими правилами:

```
UDT_definition ::= CREATE TYPE UDT_name
[subtype_clause]
[AS representation]
[instantiable_clause]
finality
[reference_type_specification]
[ref_cast_option]
[cast_option]
[method_specification_commalist]
```

Имя определяемого пользователем типа данных имеет, в общем случае, традиционную для SQL трехзвенную структуру – *имя\_каталога.имя\_схемы.имя\_типа*. Раздел подтипизации задается в следующем синтаксисе:

```
subtype_clause ::= UNDER UDT_name
```

Если этот раздел присутствует в определении *UDT*, то в нем указывается имя ранее определенного *UDT*, атрибуты и методы которого будут наследоваться определяемым *структурным типом*. *Структурные типы*, определяемые без использования наследования, называются **максимальными супертипами** (поскольку у любого из таких типов супертип отсутствует)<sup>6</sup>. В определениях *максимального структурного супертипа* или *индивидуального типа* должен присутствовать раздел *представления* (AS):

```
representation ::= predefined_type
| (attribute_definition_commalist)
```

Если в разделе *представления* указывается имя *предопределенного встроенного типа*, то определяется *индивидуальный тип*. Указание списка определений атрибутов соответствует определению *структурного типа*. Заметим, что раздел *представления* может отсутствовать. В этом случае должен присутствовать раздел подтипизации, и *представление* заново определяемого *структурного типа* полностью наследуется из определения *структурного UDT*, имя которого указано после ключевого слова **UNDER**.

## Определение атрибута структурного UDT

Определение атрибута имеет следующий синтаксис:

```
attribute_definition ::= attribute_name data_type
 [reference_scope_check]
 [default_clause]
 [collate_clause]
```

Имя определяемого атрибута должно отличаться от имен всех других атрибутов определяемого типа, включая имена атрибутов, наследуемых от *супертипа*, и имена атрибутов типа данных определяемого атрибута. Тип данных может быть любым допустимым в SQL типом данных (включая конструируемые типы **ARRAY**<sup>7</sup> и **ROW**, а также **UDT**), кроме самого определяемого *структурного типа* и его супертипов<sup>8</sup>.

Для атрибута можно объявить значение по умолчанию. Если типом данных атрибута является *встроенный тип данных*, то значение атрибута объявляется в том же синтаксисе, что и значение столбца по умолчанию в определении таблицы (см. лекцию 12). Если типом данных атрибута является **UDT** (*индивидуальный* или *структурный*), тип **ROW** или *ссылочный тип* (см. следующий пункт), то единственным допустимым значением по умолчанию является *неопределенное значение* (**NULL**). Если же типом данных атрибута является тип **ARRAY**, то значением по умолчанию может быть **NULL** или пустое значение-массив (указывается как **ARRAY[]**).

Для каждого определения атрибута, в котором типом атрибута является *структурный тип*, система автоматически генерирует пару методов, имена которых совпадают с именем атрибута. Первый метод является *наблюдателем* (*observer*). Он вызывается без явных параметров и выдает значение указанного атрибута в значении того *структурного типа*, к которому применяется. Второй метод является *мутатором* (*mutator*). Он вызывается с одним явным параметром – значением типа атрибута, применяется к некоторому местоположению (столбцу, переменной или параметру), где находится значение определяемого *структурного типа*, и этот вызов приводит к тому, что значение заменяется новым значением того же типа с измененным соответствующим образом значением данного атрибута.

Присутствие в определении атрибута раздела **reference\_scope\_check** возможно (и требуется) в том и только в том случае, когда типом определяемого атрибута является *ссылочный тип*. Более подробно мы обсудим суть этой спецификации в следующем разделе. Пока лишь кратко заметим, что этот раздел указывает системе, должна ли она проверять, что каждое значение данного атрибута является ссылкой на существующий экземпляр указанного *структурного типа*, и должна ли система вызывать *ссылочное действие* при удалении экземпляра, на который ведет ссылка<sup>9</sup>.

Можно определить *инстанцируемый* (*instantiable*) или *неинстанцируемый* (*not instantiable*) *структурный тип*:

```
instantiable_clause ::= INSTANTIABLE
 | NOT INSTANTIABLE
```

Для *неинстанцируемого типа* конструктор не определяется, и поэтому создать значение этого типа невозможно<sup>10</sup>. Такие типы применимы только для определения инстанцируемых *подтипов*. Назначение *неинстанцируемых типов* состоит в моделировании абстрактных концепций, на которых основываются более конкретные концепции. *Неинстанцируемые типы* могут быть типами атрибутов других *структурных типов*, типами столбцов, переменных и т. д. Однако в

соответствующем местоположении всегда должно находиться либо значение инстанцируемого *подтипа* данного *неинстанцируемого типа*, либо *неопределенное значение*. При отсутствии явной спецификации по умолчанию тип считается инстанцируемым.

Обязательный раздел **finality** указывает на возможность или невозможность *определения подтипов* определяемого *структурного типа*:

**finality ::= FINAL | NOT FINAL**

При определении *индивидуального типа* всегда требуется указывать **FINAL**. При определении *структурного типа* в SQL:1999 необходимо указать **NOT FINAL**. Это требование не обосновано, и в следующих версиях стандарта SQL будет разрешено определять *структурные типы*, от которых невозможно наследование.

### Раздел спецификации ссылочного типа

Хотя *типизированные таблицы* обсуждаются в следующем разделе, мы вынуждены немного забежать вперед, чтобы ввести синтаксис и пояснить смысл раздела **reference\_type\_specification** определения *структурного типа*. Строки *типизированных таблиц* обладают всеми характеристиками объектов в объектно-ориентированных системах, включая уникальные идентификаторы, которые могут использоваться для ссылок из других компонентов среды. В SQL:1999 поддерживаются три различных механизма присваивания уникальных идентификаторов экземплярам *структурных типов*, ассоциированных с такими таблицами (для всех строк таблицы, ассоциированной с данным *структурным типом*, используется один и тот же механизм). Уникальные идентификаторы экземпляров *структурного типа* могут представлять собой следующее:

- значения, генерируемые системой автоматически ( **system\_generated\_representation** );
- значения некоторого *встроенного типа* SQL, которые должны генерироваться приложением при сохранении экземпляра *структурного типа* как строки *типизированной таблицы* ( **user\_generated\_representation** );
- значения, порождаемые из одного или нескольких атрибутов *структурного типа* ( **derived\_representation** ).

Соответственно, синтаксис раздела **reference\_type\_specification** определяется следующими правилами:

**reference\_type\_specification ::= system\_generated\_representation  
| user\_defined\_representation  
| derived\_representation**

**system\_generated\_representation ::= REF IS SYSTEM GENERATED**

**user\_defined\_representation ::= REF USING predefined\_type**

**derived\_representation ::= REF USING (commalist\_of\_attributes)**

Раздел **reference\_type\_specification** может присутствовать только в определении *максимального структурного супертипа*, т. е. соответствующая спецификация наследуется всеми *подтипами* этого *супертипа*. При отсутствии в определении *супертипа* явного раздела **reference\_type\_specification** по умолчанию предполагается наличие раздела **REF IS SYSTEM GENERATED**.

## Разделы спецификации функций явного преобразования типов

Если в определении *структурного типа* присутствует раздел **reference\_type\_specification** и он имеет вид **user\_generated\_representation**, то в определении *структурного типа* должен присутствовать и раздел **ref\_cast\_option** (тем самым, раздел **ref\_cast\_option** может присутствовать только в определении *максимального структурного супертипа*). Спецификации этого раздела используются для преобразования предоставленных приложением значений *встроенного типа* в значения типа **REFERENCE ( REF )**, необходимые для реального выполнения ссылок на строки *типизированной таблицы*, и *обратного преобразования*. Синтаксис раздела определяется следующими правилами (подробнее см. в следующем разделе):

```
ref_cast_option ::= cast_to_ref
 | cast_to_type
cast_to_ref ::= CAST (SOURCE AS REF) WITH identifier
cast_to_type ::= CAST (REF AS SOURCE) WITH identifier
```

Раздел **cast\_option** может присутствовать только в определении *индивидуального типа*. Спецификации раздела обеспечивают возможности преобразования значений *индивидуального типа* в значения *базового встроенного типа*, и наоборот. Раздел имеет следующий синтаксис:

```
cast_option ::= cast_to_distinct
 | cast_to_source
cast_to_distinct ::= CAST (SOURCE_TO_DISTINCT) WITH identifier
cast_to_source ::= CAST (DISTINCT_TO_SOURCE) WITH identifier
```

## Раздел объявления сигнатур методов

В разделе **method\_specification\_commalist** объявляются сигнатуры методов, ассоциируемых с определяемым *структурным типом*. Раздел определяется следующими синтаксическими правилами:

```
method_specification ::= original_method_specification
 | overriding_method_specification
 | static_field_method_specification
original_method_specification ::= partial_method_specification
 [SELF AS RESULT]
 [SELF AS LOCATOR]
 [method_characteristic_list]
overriding_method_specification ::=
 OVERRIDING partial_method_specification
partial_method_specification ::=
 [INSTANCE | STATIC | CONSTRUCTOR]
 METHOD method_name
 SQL_parameter_declaration_list
 return_clause
 [SPECIFIC specific_method_name]
method_characteristic ::= language_clause
 | parameter_style_clause
```

```

| deterministic_clause
| SQL_data_access_indication
| null_call_clause
specific_method_name ::= [schema_name .]
qualified_identifier
static_field_method_specification ::=
 STATIC METHOD method_name ()
 RETURNS data_type
 [SPECIFIC specific_method_name]
 external_variable_name character_string_literal

```

Как показывает синтаксис, имеются возможности определять *первичные методы* ( *original\_method\_specification* ), неприменимые к любому *супертипу* определяемого *структурного типа*. Если определяемый тип является *подтипом* некоторого другого типа, то можно также определить *подменяющие методы* ( *overriding\_method\_specification* ). *Подменяющий метод* имеет то же имя и тот же *список аргументов*, что и метод, определенный в некотором *супертипе* определяемого типа.

Исходный метод может быть определен как *метод экземпляра* ( *INSTANCE* ), *статический метод* ( *STATIC* ) или *метод-конструктор* ( *CONSTRUCTOR* ). Методы экземпляра действуют над экземплярами определяемого типа. *Статические методы* не используют экземпляры типа и не влияют на них; такие методы действуют над самим типом. Наконец, *методы-конструкторы* используются для *инициализации* экземпляров типа. Поскольку у *неинстанцируемого типа* не может быть экземпляров, для него могут быть определены только *статические методы*. Если при определении *первичного метода* его разновидность не указывается, этот метод считается *методом экземпляра*.

В *сигнатуре метода* указывается имя, по которому этот метод будет вызываться ( *вызывное имя* – *invocable name* ). Кроме того, можно указать *точное имя метода* ( *specific name* ), которое может использоваться для уникальной идентификации метода, если его *вызывное имя* перегружено. Если у метода имеются какие-либо параметры, отличные от неявного параметра *SELF*, то в определении должен присутствовать заключенный в скобки список пар *<имя\_параметра, тип\_параметра>*, разделяемых запятыми. Поскольку методы являются функциями, требуется указать тип возвращаемого значения. Методы могут возвращать значения любого допустимого в SQL типа, даже *структурного типа*, ассоциированного с методом.

Наконец, у каждого метода имеется набор характеристик метода ( *method\_characteristic* ). Методы могут быть написаны на языке SQL (более точно, на SQL/PSM) или на любом из языков программирования, поддержка которых предусмотрена в стандарте SQL (*Ada*, *C/C++*, *COBOL*, *Fortran*, *MUMPS*<sup>11</sup>, *Pascal*, *PL/1*). Язык Java поддерживается в стандарте в несколько иной манере, чем другие языки. Список параметров метода может быть определен в стиле, более соответствующем стилю SQL-подпрограмм (каждый параметр может принимать неопределенное значение, и не требуется параметр *кода возврата*). Для этого в качестве характеристики метода нужно указать *PARAMETER STYLE SQL*. Можно определить список параметров в стиле, более близком стилю различных языков программирования (к параметру, который может принимать *неопределенное значение*, должен быть добавлен дополнительный параметр-индикатор, и должен быть явно определен *выходной параметр* кода ответа). В этом случае метод должен иметь

характеристику **PARAMETERSTYLE GENERAL**. Наконец, для методов, тела которых будут написаны на языке Java, нужно указать характеристику **PARAMETER STYLE JAVA**.<sup>12</sup>

Любой метод может быть *детерминированным* (**DETERMINISTIC**) или *недетерминированным* (**NOT DETERMINISTIC**). Детерминированный метод всегда возвращает один и тот же результат, если вызывается с одним и тем же набором аргументов при одном и том же состоянии базы данных. По умолчанию методы считаются недетерминированными.

У каждого метода имеется характеристика, указывающая связь этого метода с SQL. Можно указать следующие варианты:

- метод не содержит *операторов SQL* (**NO SQL**);
- метод содержит *операторы SQL*, но не обращается к базе данных (**CONTAINS SQL**);
- метод может производить выборку из базы данных, но не обновляет базу данных (**READS SQL DATA**);
- в методе допускаются обновления базы данных (**MODIFIES SQL DATA**).

По умолчанию принимается характеристика **CONTAINS SQL**. Наконец, для каждого метода можно определить его реакцию на аргументы, являющиеся *неопределенными значениями*. Если указывается **RETURN NULL ON NULL INPUT**, то метод всегда возвращает *неопределенное значение*, если значение любого из его аргументов является неопределенным (независимо от того, что написано в теле функции, реализующей метод). Если же указывается **CALLED ON NULL INPUT** (или если характеристика явно не задана), то метод всегда явно выполняется (т. е. происходит вызов соответствующей функции) при вызове с любым набором аргументов.

## Типизированные таблицы

В предыдущем подразделе уже упоминалась возможность определения *типизированных таблиц*, основанных на некотором *структурном типе*. Далее мы приведем и поясним соответствующие синтаксические правила, введем понятие иерархии *типизированных таблиц* и связь этой иерархии с иерархией *структурных типов*, а также обсудим соотношение понятия строки *типизированной таблицы* с понятием объекта в ООБД.

### Определение типизированной таблицы

С точки зрения синтаксиса оператор определения *типизированной таблицы* является частным случаем оператора создания *базовой таблицы* **CREATE TABLE**, обсуждавшегося в лекции 12 (там мы не имели возможности рассматривать этот частный случай). *Типизированные таблицы* определяются в следующем синтаксисе:

```
typed_table_defintion ::= CREATE TABLE typed_table_name
 OF UDT_name
 [UNDER typed_table_name]
 [(typed_table_element_list)]
```

Первой существенной особенностью оператора создания *типизированной таблицы* является обязательное наличие раздела **OF**, в котором указывается имя ранее

определенного *структурного типа*. Строки *типизированной таблицы* являются экземплярами ассоциированного с таблицей *структурного типа*.

## Подтаблицы и супертаблицы

Далее, при определении *типизированной таблицы* можно объявить ее *подтаблицей* некоторой другой *типизированной таблицы* (имя *супертаблицы* указывается в разделе **UNDER**). Таблица **R'** является собственной *подтаблицей супертаблицы R*, если **R'** не совпадает с **R** (в этом случае таблица **R** является собственной *супертаблицей подтаблицы R'*). Супертаблица должна быть ассоциирована со *структурным типом*, являющимся *непосредственным супертипом*<sup>13</sup> определяемой *подтаблицы*. Каждый столбец указанной *супертаблицы* наследуется *подтаблицей*; наследуются и характеристики столбцов *супертаблицы* – значения по умолчанию, ограничения *целостности* и т. д. Эти столбцы называются *унаследованными столбцами подтаблицы*, и они соответствуют атрибутам *UDT подтаблицы*, унаследованным от *UDT супертаблицы*. Кроме того, *подтаблица* будет содержать по одному столбцу для каждого собственного атрибута ассоциированного *структурного типа*. Такие столбцы *подтаблицы* называются заново определенными.

Как это принято в SQL, столбцы *типизированной таблицы* имеют порядковые номера. При этом унаследованные столбцы нумеруются до заново *определенных столбцов* и имеют те же номера, которые имели столбцы *супертаблицы*.

## Определение элементов типизированной таблицы

Заключительным компонентом определения *типизированной таблицы* является конструкция **typed\_table\_element\_list**, являющаяся обобщением конструкции **table\_element\_list**, которая используется в определении обычной *базовой таблицы* (см. лекцию 12). Элемент списка элементов *типизированной таблицы* определяется следующим синтаксическим правилом:

```
type_table_element ::= table_constraint_definition
 | self-referencing_column_specification
 | column_options
```

Как видно из этого правила, в определении *типизированной таблицы* разрешается указывать табличные *ограничения целостности*. Если определяемая таблица является *подтаблицей* некоторой *супертаблицы*, то в ней не допускается определение *ограничения первичного ключа (PRIMARY KEY)*. Однако если определяется *максимальная супертаблица*, то в ее определении допускается спецификация **PRIMARY KEY** (с указанием одного или нескольких столбцов) или спецификация ограничения **UNIQUE** (с указанием одного или нескольких столбцов) в комбинации с указанием **NOT NULL**. В определении *типизированной таблицы* могут также содержаться спецификации *ссылочных ограничений целостности*. Ссылки могут вести как на *типизированную*, так и на обычную таблицу.

"Самоссылающийся" (self-referencing) столбец специфицируется в следующем синтаксисе:

```
REF IS column_name { SYSTEM GENERATED
 | USER GENERATED | DERIVED }
```

Эта спецификация не может входить в определение *подтаблицы*. Спецификация должна присутствовать в определении максимальной *супертаблицы*, и самоссылающийся столбец, определенный в максимальной *супертаблице*, наследуется любой ее *подтаблицей*. Семантика самоссылающихся столбцов обсуждается в следующем пункте.

Последней разновидностью элемента *типизированной таблицы* являются опции столбцов (**column\_options**). Опции столбца можно указывать только для заново *определенных столбцов* — для унаследованных столбцов это не допускается. Соответствующая конструкция имеет следующий синтаксис:

```
column_name WITH OPTIONS ::= scope_clause
| default_clause
| column_constraint_definition_list
| collate_clause
```

Раздел **scope\_clause** может входить в опции только заново определяемого столбца с типом **REF** (подробности в следующем подразделе). Для заново определяемого столбца некоторого *типа символьных строк* можно указать раздел **collate\_clause**, чтобы задать желаемый порядок на соответствующем наборе символов. Если требуется указать значение столбца по умолчанию, отличное от значения по умолчанию соответствующего атрибута, ассоциированного с определяемой таблицей *структурного типа*, можно воспользоваться опцией **default\_clause**. Наконец, для заново определяемого столбца можно указать одно или несколько ограничений, включая *проверочные ограничения* (см. лекцию 12).

## Ссылочные значения и REF-типы

Понятия *ссылочных значений* и *ссылочных (REF) типов* являются, по существу, неразделимыми. В SQL:1999 *ссылочный тип* может использоваться в качестве *типа данных столбцов* обычных таблиц, атрибутов *структурных типов*, SQL-переменных и параметров – словом, везде, где можно использовать другие типы данных SQL. Значения местоположения *ссылочного типа* всегда являются *ссылочными значениями* строк *типизированных таблиц* (т. е. значениями самоссылающихся столбцов этих строк).

Для удобства повторим синтаксис спецификации *ссылочного типа*:

```
reference_type_specification ::= system_generated_representation
| user_defined_representation
| derived_representation
system_generated_representation ::= REF IS SYSTEM GENERATED
user_defined_representation ::= REF USING predefined_type
derived_representation ::= REF USING (commalist_of_attributes)
Механизмы генерации ссылочных значений
```

В SQL:1999 и SQL:2003 обеспечиваются три механизма назначения уникальных идентификаторов экземплярам *структурных типов*, ассоциированных с *типизированными таблицами*. Во всех *типизированных таблицах*, ассоциированных с данным *структурным типом*, должен использоваться один и тот же механизм.

Предоставляются следующие альтернативы выбора *ссылочных значений*, которые могут являться:

- значениями некоторого *встроенного типа* SQL (**user\_defined\_representation**), которые должны генерироваться приложением каждый раз при сохранении экземпляра *структурного типа* как строки *типизированной таблицы* ;
- значениями, порождаемыми из одного или нескольких атрибутов *структурного типа* ;
- значениями, автоматически генерируемыми системой.

Как отмечалось в разделе "Определяемые пользователями типы", при определении любого *максимального структурного супертипа* явно или неявно задается спецификация *ссылочного типа*. Спецификация *ссылочного типа* наследуется всеми *подтипами* этого *супертипа*. При определении *типизированных таблиц* необходимо указать соответствующую спецификацию самоссылающегося столбца (конечно, эта спецификация логически избыточна, и, по всей вероятности, в следующих версиях стандарта SQL это требование будет ослаблено). Хотя соотношение между альтернативами спецификации *ссылочного типа* и спецификации самоссылающегося столбца очевидно, приведем его явно ([табл. 19.1](#)).

Таблица 19.1. Спецификации *ссылочного типа* и самоссылающегося столбца

| <b>reference_type_specification</b>     | <b>self-referencing_column</b> |
|-----------------------------------------|--------------------------------|
| <b>REF USING predefined_type</b>        | <b>USER GENERATED</b>          |
| <b>REF FROM commalist_of_attributes</b> | <b>DERIVED</b>                 |
| <b>REF IS SYSTEM GENERATED SYSTEM</b>   | <b>GENERATED</b>               |

Если для некоторого *структурного типа* выбран вариант пользовательской генерации *ссылочных значений*, то ответственность за поддержание уникальности таких значений лежит на пользователе. Конечно, ограничения **PRIMARY KEY** или **UNIQUE**, определенные на уровне максимальной *супертаблицы* семейства *типизированных таблиц*, могут гарантировать отсутствие в любой таблице этого семейства дублирующих *ссылочных значений*, но в SQL:1999 отсутствуют какие-либо средства, предотвращающие повторное использование *ссылочных значений* из удаленных строк в самоссылающихся столбцах новых строк.

### Преобразование задаваемых пользователем *ссылочных значений* к *ссылочному типу*

В этом случае в определении *структурного типа* может присутствовать конструкция **ref\_cast\_option** (вернее, она должна присутствовать в определении соответствующего максимального *супертипа*). Синтаксис этой конструкции приводился в предыдущем разделе, но для удобства мы его повторим здесь:

```

ref_cast_option ::= cast_to_ref
 | cast_to_type
cast_to_ref ::= CAST (SOURCE AS REF) WITH identifier
cast_to_type ::= CAST (REF AS SOURCE) WITH identifier

```

Чтобы пояснить эту конструкцию, предположим, что в определении *структурного типа* указано **REF USING INTEGER**. Тогда соответствующие приложения отвечают за то, чтобы обеспечить глобально уникальные целые значения самоссылающегося столбца во всех строках всех *типизированных таблиц*, ассоциированных с этим *структурным типом*. Но приложения обеспечивают значения *целого типа*, а типом данных самоссылающегося столбца является некоторый *ссылочный тип*.

Для решения именно этой проблемы и предназначена конструкция **ref\_cast\_option**. В этой конструкции вводятся имена двух SQL-функций, первая из которых служит для преобразования *ссылочных значений*, обеспечиваемых приложением, к соответствующему **REF**-типу при вставке или обновлении строк *типизированной таблицы* (**SOURCE AS REF**). Вторая функция преобразует значения **REF**-типа к соответствующему встроенному типу данных при выборке строк из *типизированной таблицы* (**REF AS SOURCE**). Система автоматически генерирует обе подпрограммы, и конструкция **ref\_cast\_option** позволяет лишь назначить подпрограммам имена. Если конструкция **ref\_cast\_option** явно не включается в определение *структурного типа* с **REF USING predefined\_type**, то имена подпрограмм назначаются системой. Единственным преимуществом явного назначения имен является возможность явного вызова этих функций при написании SQL-операторов, содержащих выражения **REF**-типа, которые нужно привести к соответствующему *встроенному типу*. Заметим, что такие функции невозможно написать вручную, поскольку правила отображения зависят от реализации SQL.

Если для *структурного типа* выбирается альтернатива порождения *ссылочных значений*, то система использует для порождения *ссылочных значений* значения неявно указанных столбцов (соответствующих явно указанным атрибутам ассоциированного *структурного типа*). При этом остаются все упомянутые выше проблемы, хотя в таком случае явно требуется объявление ограничений **PRIMARY KEY** или **UNIQUE** для соответствующего набора столбцов.

Наконец, при выборе последней альтернативы (системно-генерируемые *ссылочные значения*) каждой строке, вставляемой в *типизированную таблицу*, ассоциированную с соответствующим *структурным типом*, присваивается уникальный идентификатор. Это значение сохраняется в самоссылающемся столбце и может быть использовано любым приложением для уникальной идентификации данной строки на всем протяжении жизни таблицы.

## Спецификация ссылочного типа при объявлении столбцов и атрибутов

Самоссылающиеся столбцы всегда имеют **REF**-тип. Конкретный **REF**-тип зависит от двух факторов:

- *структурного типа*, ассоциированного с *типизированной таблицей*: **REF**-тип всегда связан с некоторым *структурным типом*;
- выбранного способа генерации *ссылочных значений*; эта информация задается в определении *структурного типа* и не присутствует в спецификации *ссылочного типа*.

Для объявления местоположения *ссылочного типа* используется следующий синтаксис:

```
reference_type ::= REF (referenced_type) [SCOPE table_name]
referenced_type ::= UDT_name
```

**UDT\_name** должно задавать имя типа ( **referenced\_type** ), на экземпляры которого будут указывать значения *ссылочного типа*. **REF** - тип может использоваться в качестве типа атрибута *структурного типа*, и в этом случае **referenced\_type** может быть тем же самым, что и определяемый *структурный тип*. Во всех остальных случаях **referenced\_type** должен являться некоторым существующим *структурным типом*.

В необязательном разделе **SCOPE** задается имя *типизированной таблицы*. Ассоциированным *структурным типом* этой таблицы должен быть **referenced\_type REF** - типа, в спецификации которого содержится данный раздел **SCOPE**. Хотя можно было бы ожидать, что значение **REF** - типа можно использовать для ссылки на строки *типизированных таблиц*, ассоциированный *структурный тип* которых является собственным *подтипом* указанного **referenced\_type**, в SQL такая разновидность ссылок не допускается. Ассоциированный *структурный тип* таблицы, на строки которой указывают значения **REF** - типа, должен быть в точности тем же, что и **referenced\_type** этого **REF** - типа. Но, конечно, можно объявить **REF** - тип, у которого **referenced\_type** является ассоциированным *структурным типом подтаблицы*, хотя самоссылающийся столбец этой *подтаблицы* необходимо наследуется от максимальной *супертаблицы* семейства таблиц.

### Поддержка согласованности ссылок

Никакое *ссылочное значение* никогда не идентифицирует какую-либо строку, кроме той, с которой оно было ассоциировано с самого начала. Если эта строка удаляется, то значение ничего не идентифицирует и никогда не может быть связано с другой строкой. Из этого следует, что система должна каким-либо образом узнавать о том, идентифицирует ли данное *ссылочное значение* какую-то хранимую строку или ничего не идентифицирует (является висящей ссылкой). Но как система может это узнать, не потратив множество ресурсов? Отчасти здесь может помочь раздел **SCOPE**. В этом разделе указывается одна таблица, в которой строки должны существовать для всех значений данного местоположения, типом данных которого является некоторый **REF** - тип. (В будущих версиях стандарта SQL, по всей видимости, будет разрешено указывать в разделе **SCOPE** список имен *типизированных таблиц* или даже использовать некоторую конструкцию, означающую "все таблицы, ассоциированные с данным *структурным типом*".)

Итак, если определяется столбец таблицы, поле строчного типа или атрибут *структурного типа*, и типом этого местоположения является **REF** - тип, то можно специфицировать раздел **SCOPE**. Однако если такой раздел действительно указывается, то требуется также указать, нужна ли проверка *ссылочных значений*. Для этого служит конструкция **reference\_scope\_check**, определяемая следующим синтаксическим правилом:

```
reference_scope_check ::= REFERENCES ARE [NOT] CHECKED
[ON DELETE referential_action]
```

Если указывается **REFERENCES ARE NOT CHECKED** или если раздел **SCOPE** не задается, то в определяемом местоположении можно хранить любое *ссылочное значение*, независимо от того, является ли оно значением самоссылающегося столбца какой-либо таблицы, на строку которой предположительно указывает ссылка. В этом случае система не гарантирует, что *ссылочное значение* действительно указывает на строку (но, конечно, это значение должно быть значением правильного типа – **REF**-типа указанного *структурного типа* ).

Если же указывается **REFERENCES ARE CHECKED**, то каждый раз при сохранении значения в определяемом столбце, поле или атрибуте система обращается к указанной в разделе **SCOPE** таблице, чтобы убедиться в том, что в ней имеется строка, значение самоссылающегося столбца которой совпадает с сохраняемым *ссылочным значением*. Кроме того, если указывается **REFERENCES ARE CHECKED**, то можно также указать *ссылочное действие*, которое должно выполняться при удалении строки, идентифицируемой *ссылочным значением*. Как обычно (см. лекцию 12), возможными *ссылочными действиями* являются **RESTRICT**, **CASCADE**, **SET NULL** и **NO ACTION**. Если *ссылочное действие* явно не указывается, по умолчанию принимается **NO ACTION**. (Для поля строчного типа (**ROW TYPE**) и атрибута *структурного типа* допускается только **NO ACTION**.)

Заметим, что если раздел **SCOPE** включается в определение атрибута *структурного типа*, то в конструкции **column\_options** столбца *типизированной таблицы*, соответствующего данному атрибуту, раздел **SCOPE** присутствовать не может – это считается *синтаксической ошибкой*.

### Выборка данных из типизированных таблиц

Приведем несколько примеров операций выборки данных из *типизированных таблиц*, а также кратко обсудим операции обновления таких таблиц. Для этого сначала определим *структурные типы* **EMP\_T**, **PROGRAMMER\_T** и **DEPT\_T**, а также соответствующие *типизированные таблицы* (упрощенный вариант).

```
CREATE TYPE EMP_T AS (
 EMP_NAME VARCHAR(20),
 EMP_BDATE DATE,
 EMP_SAL SALARY,
 DEPT REF (DEPT))
INSTANTIABLE
NOT FINAL
REF IS SYSTEM GENERATED
INSTANCE METHOD age ()
RETURNS DECIMAL (3,1);
CREATE TYPE PROGRAMMER_T UNDER EMP_T AS (
 PROG_LANG VARCHAR (10))
INSTANTIABLE
NOT FINAL;
CREATE TYPE DEPT_T AS (
 DEPT_NO INTEGER,
 DEPT_NAME VARCHAR(200),
 DEPT_MNG REF (EMP))
```

```

INSTANTIABLE
REF IS SYSTEM GENERATED
NOT FINAL;
CREATE TABLE EMP OF EMP_T
(REF IS DEPT_ID SYSTEM GENERATED,
DEPT WITH OPTIONS SCOPE DEPT);
CREATE TABLE PROGRAMMER OF PROGRAMMER_T UNDER EMP;
CREATE TABLE DEPT OF DEPT_T
(REF IS EMP_ID SYSTEM GENERATED,
DEPT_MNG WITH OPTIONS SCOPE EMP);

```

Следует отметить, что с *типизированными таблицами* можно работать, как с обычными таблицами<sup>14</sup>. Поэтому, в частности, возможен следующий запрос.

```

SELECT EMP_NAME
FROM EMP
WHERE EMP_SAL < 20000.00;

```

Пример 19.1. Найти имена всех служащих, размер заработной платы которых меньше 20000.00.

В соответствии с семантикой SQL:1999, при выполнении запроса из [примера 19.1](#) сначала будет произведена выборка имен служащих, удовлетворяющих условию, из таблицы *EMP*, затем – из таблицы *PROGRAMMER*, и эти промежуточные результаты будут скомбинированы в окончательный результат путем применения операции объединения (*UNION*). Но предположим, что нас интересуют только те служащие, получающие зарплату, не превышающую 20000 руб., которые не являются программистами ([пример 19.2](#)). Тогда можно применить формулировку запроса, в которой присутствует спецификация *ONLY*:

```

SELECT EMP_NAME
FROM ONLY (EMP)
WHERE EMP_SAL < 20000.00;

```

Пример 19.2. Найти имена всех служащих, которые не являются программистами, размер заработной платы которых меньше 20000.00.

Естественно, в запросах к *типизированным таблицам* можно использовать ссылки.

```

SELECT EMP_NAME, DEPT -> DEPT_NAME
FROM EMP
WHERE EMP_SAL < 20000.00;

```

Пример 19.3. Найти имена и названия отделов, где работают служащие, размер заработной платы которых меньше 20000.00.

В SQL:1999 операция "*.->*" называется операцией *разыменования (dereferencing)*, но в обиходе ее можно считать операцией перехода по ссылке (в нашем примере *DEPT* ссылается на *DEPT\_NAME*). Можно неформально трактовать *ссылочные значения* как указатели на строки *типизированных таблиц*.

Может показаться неожиданным, что запрос из [примера 19.3](#) выбирает значения из таблицы **DEPT**, хотя в разделе **FROM** этого запроса она даже не упоминается. Дело в том, что выполнение операции *разыменования* фактически приводит к выполнению соединения таблиц **EMP** и **DEPT**, делая в запросе столбец **DEPT\_NAME** "видимым".

Конечно, в запросе допускаются многократные переходы по ссылкам, так что можно сформулировать следующий запрос:

```
SELECT EMP_NAME, DEPT -> DEPT_MNG -> EMP_NAME
FROM EMP
WHERE EMP_SAL < 20000.00;
```

Пример 19.4. Найти имена служащих и имена руководителей их отделов для служащих, получающих зарплату, не превышающую 20000.00.

Как показывает следующий пример, в запросах можно использовать вызовы методов над строками, к которым производится переход по ссылке.

```
SELECT DEPT_MNG -> EMP_NAME, DEPT_MNG -> age ()
FROM DEPT
WHERE DEPT_NO = 605;
```

Пример 19.5. Найти имя и возраст руководителя отдела 605.

Наконец, имеется возможность полностью выбрать экземпляр *структурного типа*, идентифицируемый *ссылочным значением* (в SQL:1999 это называется разрешением ссылки – *reference resolution*).

```
SELECT Deref (DEPT_MNG)
FROM DEPT
WHERE DEPT_NO = 605;
```

Пример 19.6. Получить полные данные о руководителе отдела 605.

В этом случае результатом запроса будет являться таблица, включающая один столбец *структурного типа* **EMP\_T**. Единственным значением этого столбца будет экземпляр (значение) этого *структурного типа*, соответствующий служащему-руководителю отдела 605.

Операции обновления *типизированных таблиц* выполняются очевидным образом. Операция **INSERT** вставляет указанные строки в указанную *таблицу*. Операции **DELETE** и **UPDATE** удаляют или модифицируют строки в иерархии таблиц, корнем которой является указанная таблица, если в операции не содержится **ONLY**. Если же специфицировано **ONLY**, то удаляются или модифицируются только строки указанной таблицы.

## Типизированные представления

Наряду с *типизированными базовыми таблицами* в SQL:1999 поддерживаются *типизированные представления*, иначе называемые *представлениями*, на которые можно ссылаться (*referenceable views*). Иногда такие *представления* также называют *объектными представлениями*, поскольку данные, видимые через *представление*, соответствуют строкам *типизированных таблиц*, поведение которых

во многом похоже на поведение объектов в объектно-ориентированных системах. Между *типизированными базовыми таблицами* и *типизированными представлениями* имеется большое сходство, но есть и несколько отличий, связанных с различиями базовых таблиц и *представлений*.

В SQL в связи с *объектными представлениями* вводится ряд терминов – "*суперпредставление*", "*подпредставление*", "*непосредственное суперпредставление*", "*непосредственное подпредставление*", "*собственное суперпредставление*" и "*собственное подпредставление*". Смысл этих терминов полностью аналогичен смыслу соответствующих терминов для *типизированных базовых таблиц*. Термин *семейство подтаблиц* применяется как к *типизированным таблицам*, так и к *типизированным представлениям*.

Определение *типизированного представления* задается в следующей синтаксической форме:

```
view_definition ::= CREATE VIEW table_name
 OF UDT_name
 UNDER table_name
 (view_element_commalist)
 AS query_expression
 [WITH [levels_clause] CHECK OPTION]
view_element ::= self_referencing_column_specification
 | column_name WITH OPTIONS scope_clause
```

Указываемое **UDT\_name** должно быть именем существующего *структурного типа*. Как и в определении обычных *представлений*, в разделе **AS** указывается выражение запроса. В случае *типизированных представлений* это выражение запроса должно основываться на единственной *типизированной таблице* (базовой таблице или *представлении*). Эта *типизированная таблица* должна быть ассоциирована с тем *жеструктурным типом*, что и определяемое *представление*. Такую таблицу иногда называют базисной таблицей *представления*.

*Типизированное представление* можно определить как *подпредставление* другого *типизированного представления*. В этом случае *структурный тип*, ассоциированный с определяемым *представлением*, должен являться непосредственным *подтипом структурного типа*, который ассоциирован с *суперпредставлением*, специфицируемым в разделе **UNDER**. Базисная таблица определяемого *представления* должна являться *собственной подтаблицей* или *собственным подпредставлением* – не обязательно непосредственным – базисной таблицы *непосредственного суперпредставления* определяемого *представления*.

В определении *типизированного представления* может входить один или несколько элементов **column\_name WITH OPTIONS scope\_clause**. Если *представление* определяется как *подпредставление* другого *типизированного представления*, то в его определении не должна содержаться спецификация самоссылающегося столбца. Если определяется *максимальное суперпредставление* (т. е. в определении не содержится раздел **UNDER**), то эта спецификация может присутствовать. Если спецификация присутствует, то она может содержать только конструкции **USER GENERATED** или **DERIVED** (из этого следует, что нельзя определить *типизированное*

*представление*, в ассоциированном *структурном типе* которого присутствует спецификация **REF IS SYSTEM GENERATED** ). При указании **USER GENERATED** степень определяемого *представления* на единицу больше числа атрибутов ассоциируемого *структурного типа* ; дополнительным столбцом является самоссылающийся столбец. В случае указания **DERIVED** дополнительный столбец не появляется, поскольку значение самоссылающегося столбца порождается из тех же столбцов, из которых порождается значение самоссылающегося столбца базисной таблицы.

## Заключение

Если обратиться к истории, выяснится, что попытки расширения функциональности *СУБД*, изначально основанных на реляционном подходе, предпринимались уже на ранних стадиях разработки таких систем. Классическими примерами являются проекты *System R* компании *IBM*, где разработчики пытались обеспечить возможности работы со сложными объектами путем расширения *SQL*, и *Ingres* (университет Беркли), где Майкл Стоунбрейкер предлагал механизм определения пользовательских типов данных на основе *представлений* и *хранимых процедур*. Однако новый толчок к расширению *SQL*-ориентированных *СУБД* объектными свойствами был получен со стороны объектного мира после публикации *Первого манифеста*.

В ответном *Втором манифесте* представители индустрии развитых *СУБД* утверждали, что имеются реальные возможности добиться желаемой функциональности без коренной ломки традиционной технологии. Идеи *Второго манифеста* были воплощены в жизнь в нескольких ведущих *SQL*-продуктах, и использование объектных расширений позволило самим поставщикам обеспечить ряд законченных функциональных расширений своих систем. Однако ожидания большого спроса со стороны пользователей на сами инструменты объектных расширений не оправдались. Некоторые известные специалисты из области баз данных считают, что для этого еще не пришло время.

Развитие объектно-реляционного подхода нашло отражение в языке *SQL*. Гигантский *стандарт SQL:1999* позволяет хотя бы сопоставлять отдельные реализации, хотя ни одна компания полностью его не поддерживает. Как можно заметить, разработчики стандарта *SQL* пошли на существенно большее сближение с *объектно-ориентированным подходом* к организации систем баз данных, чем это предполагалось во *Втором манифесте*. В особенности это проявляется в механизмах *типизированных таблиц*, *ссылочных типов* и *ссылочных значений*: *типизированные таблицы* похожи на экстенды классов, а *ссылочные значения* – на объектные идентификаторы. Однако во многом это сходство является внешним – за путевыми выражениями в стиле *ODMG* по-прежнему скрываются *операции* соединения таблиц.

Данная Лабораторная работасодержит весьма разнообразный материал, объединенный только общей идеей расширения *РСУБД* объектными возможностями. К сожалению, это вынужденное разнообразие, поскольку, на мой взгляд, большая часть расширений выполнялась без предварительной проработки не только общей модели, но даже и концепции языка. В результате мы можем оказаться в ситуации, когда язык *SQL* в лучшем случае будет полностью понятен только главному редактору стандарта.

И последнее замечание, на котором мы закончим этот курс. Несмотря на некоторую критику в *адрес* языка *SQL*, высказанную в начале лекции 11, мы потратили на обсуждение этого языка половину курса и больше его практически не критиковали. Не означает ли это, что язык все-таки очень хорош или что *автор* питает к нему особую

привязанность? Конечно же нет! В языке *SQL* имеется множество слабых мест, неточностей и даже прямых ошибок. Если задаться целью продемонстрировать все промахи языка *SQL*, то этот курс никогда бы не закончился, а его читатели так и не узнали бы, что представляет собой язык в целом.

При всех недостатках у *SQL* имеются два неоспоримых преимущества. Во-первых, за 30 лет существования языка к нему привыкли (и даже сроднились с ним) тысячи профессионалов в области баз данных. Как говорится, лучше плохое, да свое. Во-вторых (и это проверено многолетней практикой) язык *SQL* допускает эффективную и *масштабируемую* реализацию, и даже объектные расширения языка не вызывают какой-либо деградации производительности систем. Одним словом, нам предстоит еще долгая совместная жизнь с *SQL*, и, чтобы она была удачной, нужно хорошо знать и достоинства, и недостатки этого языка.