Учебные материалы для курса «СУБД Oracle»

v.1.8 09.09.2015

Составитель:
ст. преподаватель каф. ТИ
ЯрГУ им. П.Г. Демидова
Горбунов О.Е.

План курса СУБД Oracle

Введение

Курс расчитан на студентов старших курсов ВУЗов по специальностям, связанным с IT. Предполагается, что студенты уже знакомы с основами реляционных баз данных. В курсе детально разбирается реляционная СУБД компании Oracle.

Данный курс основан на нескольких источниках по базам данных и, в частности, по СУБД Oracle, указанным в списке литературы. В материале курса учтен опыт разработки и преподавания автором соответствующей дисциплины.

Терминология

Напомним основные понятия.

Предметная область - некоторая часть реально существующей системы, функционирующая как самостоятельная единица. Полная предметная область может представлять собой экономику страны, однако на практике для информационных систем наибольшее значение имеет предметная область масштаба отдельного предприятия или корпорации.

База данных (БД, database) - поименованная совокупность структурированных данных, относящихся к определенной предметной области.

Система управления базами данных (СУБД) - комплекс программных и языковых средств, необходимых для создания и модификации базы данных, добавления, модификации, удаления, поиска и отбора информации, представления информации на экране и в печатном виде, разграничения прав доступа к информации, выполнения других операций с базой.

Модель данных, или концептуальное описание предметной области - самый абстрактный уровень проектирования баз данных.

На следующем рисунке приведена схема, показывающая взаимосвязь основных терминов в области проектирования баз данных и работы с ними.

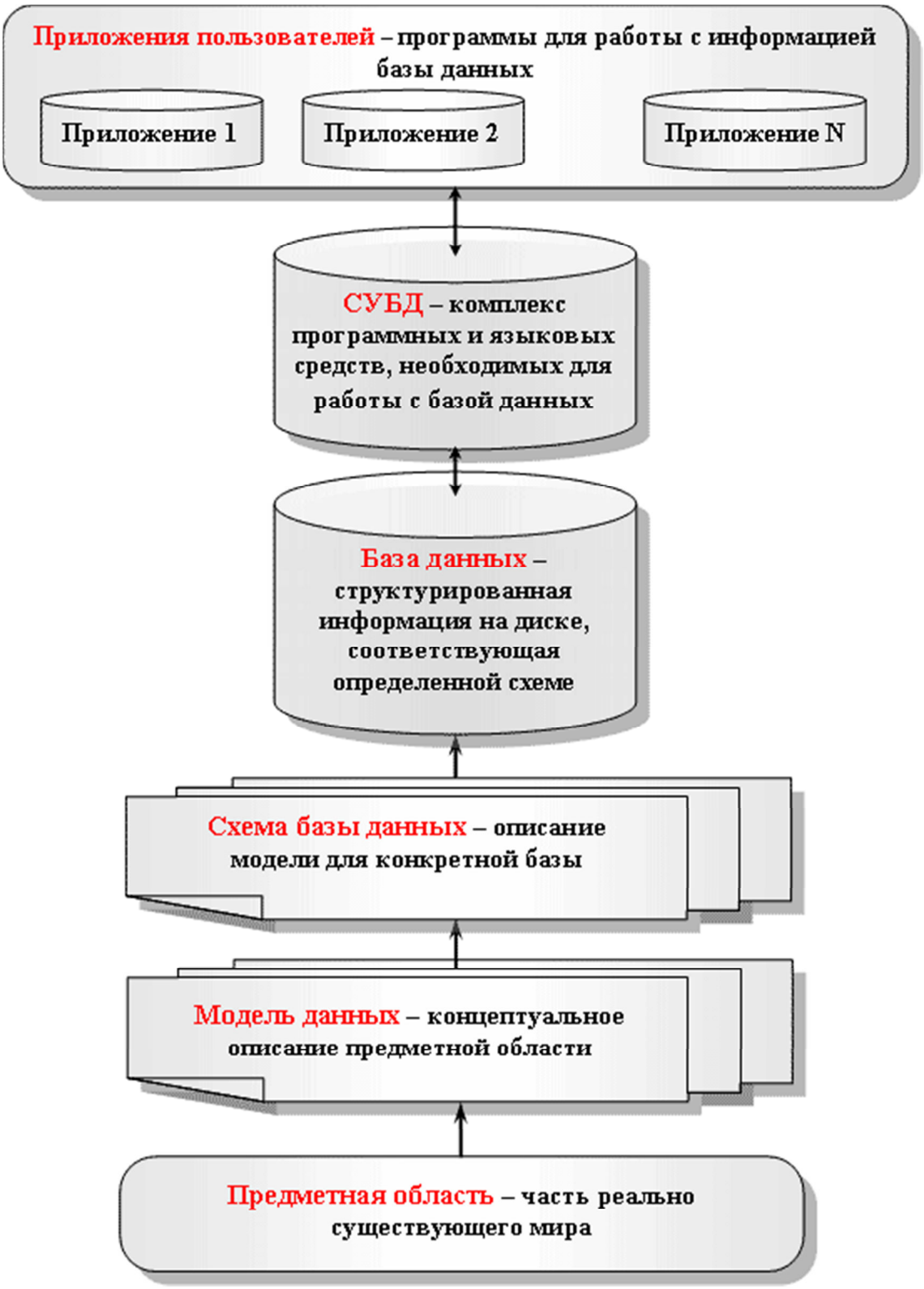


Рис. Взаимосвязь основных терминов в области проектирования баз данных и работы с ними

Мы будем рассматривать реляционные базы данных, а именно СУБД Oracle.

Реляционная БД - основной тип современных баз данных.

Edgar Frank Codd (Эдгар Франк Кодд) опубликовал работу “A Relational Model of Data for Large Shared Data Banks” в июне 1970 года в журнале ACM (Association of Computer Machinery). Модель Кодда теперь принята как базовая модель для реляционных СУБД. Язык, называемый Structured Query Language (аббревиатура SQL, произносится «sequel»), разработан IBM Corporation Inc. для использования модели Кодда. В 1979 Relational Software (теперь Oracle Corp.) представила первую коммерческую реализацию SQL. Позже SQL был стандартизован (произносится «sql») и принят как стандартный язык реляционных СУБД.

Системы с поддержкой SQL в настоящее время стали преобладающими на рынке баз данных, и одной из важных причин подобного состояния дел является именно то, что такие системы основаны на реляционной модели данных.

Достоинствами реляционного подхода принято считать следующие свойства:

- реляционный подход основывается на небольшом числе интуитивно понятных абстракций, на основе которых возможно простое моделирование наиболее распространенных предметных областей. Эти абстракции могут быть точно и формально определены;

- теоретическим базисом реляционного подхода к организации баз данных служит простой и мощный математический аппарат теории множеств и математической логики;

- реляционный подход обеспечивает возможность ненавигационного манипулирования данными без необходимости знания конкретной физической организации баз данных во внешней памяти.

Напомним, что реляционная модель распространяется на три принципиальных аспекта организации данных: структура данных, манипулирование данными и целостность данных.

Существуют и другие модели и системы управления базами данных (иерархические, сетевые, реляционные и т.д.).

Основы теории реляционных баз данных

Используемая терминология отличается с точки зрения практической работы с СУБД, с точки зрения теории реляционных баз данных и с точки зрения концептуальной модели.

Основные понятия теории реляционных баз данных: тип данных, домен, атрибут, кортеж, отношение, первичный ключ.

Понятие *типа данных* в реляционной модели данных полностью соответствует понятию типа данных в языках программирования. Традиционное (нестрогое) определение типа данных состоит из: определения множества значений данного типа; определение набора операций, применимых к значениям типа; определение способа внешнего представления значений типа (литералов).

Понятие *домена* более специфично для баз данных, хотя и имеются аналогии с подтипами в некоторых языках программирования. В общем виде домен определяется путем задания некоторого базового типа данных, к которому относятся элементы домена, и произвольного логического выражения, применяемого к элементу этого типа данных (ограничения домена). Элемент данных является элементом домена в том и только том случае, если вычисление этого логического выражения дает результат истина. С каждым доменом связывается имя, уникальное среди всех доменов базы данных.

Например, "целое число" - это тип данных, а "возраст" - это домен.

Отношение r состоит из заголовка (Hr) и тела (Br).

Заголовком отношения r, Hr, называется конечное множество упорядоченных пар <A, T>, где A – имя атрибута, а T – имя некоторого типа данных или домена. Все имена атрибутов в заголовке отношения должны быть различны.

Спецификация атрибута состоит из его названия, указания типа данных и описания ограничений целостности - множества значений (или домена), которые может принимать данный атрибут.

Кортеж tr, соответствующий заголовку Hr, - множество упорядоченных триплетов вида <A, T, v>, по одному триплету для каждого атрибута в Hr. Здесь v – допустимое значение типа данных или домена T.

Телом Br отношения r называется произвольное множетсов кортежей tr.

Значением Vr отношения r называется пара множеств Hr и Br.

Переменной VARr называется именованный контейнер, который может содержать любое допустимое значение Vr.

Степенью, или арностью, заголовка отношения, кортежа, соответствующему этому заголовку, тела отношения, значения отношения и переменной отношения является мощность заголовка отношения.

Схемой реляционной базы данных считается набор пар <имя_VARr, Hr>, включающий имена и заголовки всех переменных отношения, которые определены в базе данных. Реляционная база данных – это набор пар <VARr, Hr>.

С точки зрения практической работы с СУБД отношение обычно называется таблицей (table), схема отношения – заголовок таблицы, кортеж – строка таблицы (или запись), атрибут отношения – столбец таблицы, имя атрибута – имя столбца.

В концептуальной модели реляционной БД аналогом отношения является сущность (entity), с определенным набором свойств - атрибутов, способных принимать определенные значения (набор допустимых значений - домен).

Сущность - некоторый обособленный объект или событие, информацию о котором необходимо сохранять в базе данных, имеющий определенный набор свойств - атрибутов. Сущности могут быть как физические (реально существующие объекты: например, СТУДЕНТ, атрибуты - № зачетной книжки, фамилия, его факультет, специальность, № группы и т.д.), так и абстрактные (например, ЭКЗАМЕН, атрибуты - дисциплина, дата, преподаватель, аудитория и пр.). Для сущностей различают ее тип и экземпляр. Тип характеризуется именем и списком свойств, а экземпляр - конкретными значениями свойств.

Первичным ключем (primary key) переменной отношения является такое подмножество S атрибутов ее заголовка, что в любое время значение первичного ключа в любом кортеже тела отношения отличается от значения первичного ключа в любом другом кортеже тела этого отношения, а никакое собственное подмножество S этим свойством не обладает.

То свойство, что тело любого отношения не содержит кортежей-дубликатов, следует из определения тела отношения как множества кортежей. Из этого свойства вытекает наличие у каждого значения отношения первичного ключа – минимального множества атрибутов, являющегося подмножеством заголовка данного отношения, составное значение которых уникально определяет кортеж отношения. В концептуальной модели первичный ключ - набор атрибутов сущности, однозначно идентифицирующий экземпляр сущности.

Могут также существовать альтернативные минимальные наборы атрибутов, обладающие свойством уникальности. Они называются возможными ключами (candidate key).

Проектировщик базы данных должен для каждой таблицы указать один первичный ключ и возможные ключи. Это позволяет СУБД определять и проверять ограничения целостности (и таким образом поддерживая целостность базы данных).

Отметим, что, по определению, в отношении отсутствует упорядоченность кортежей и атрибутов.

Атомарность значений атрибутов, 1НФ.

Значения всех атрибутов являются атомарными. Это следует из определения домена как потенциального множества значений скалярного типа данных, т.е. среди значений домена не могут содержаться значения с видимой структурой. Со всеми значениями можно обращаться только с помощью операций, определенных в соответствующем типе данных.

Принято говорить, что в реляционных базах данных допускаются только нормализованные отношения, или отношения, представленные в 1НФ.

Связь (relation) - функциональная зависимость между отношениями. В реляционной модели данных между отношениями устанавливаются связи по ключам, один из которых в главной (parent, родительской) таблице - первичный, второй - внешний ключ *(foreign key)* - во внешней (child, дочерней) таблице, как правило, первичным не является и образует связь "один ко многим" (1:N). В случае первичного внешнего ключа связь между таблицами имеет тип "один к одному" (1:1).

Связи на концептуальном уровне представляют собой простые ассоциации между сущностями. Например, утверждение "Покупатели приобретают продукты" указывает, что между сущностями "Покупатели" и "Продукты" существует связь, и такие сущности называются участниками этой связи.

Существует несколько типов связей между двумя сущностями: это связи "один к одному", "один ко многим" и "многие ко многим".

Целостость сущности и ссылок

Целостность сущности: у любой переменной отношения должен существовать первичный ключ, и никакое значение первичного ключа в кортежах значения-отношения переменной отношения не должно содержать неопределенных значений.

Целостность по ссылкам: для каждого значения внешнего ключа, появляющегося в котреже значения-отношения ссылающейся переменной отношения, либо в значении-отношении переменной отношения, на которую указывает ссылка, должен найтись кортеж с таким же значением первичного ключа, либо значение внешнего ключа должно быть полностью неопределенным.

Ограничения целостности сущности и по ссылкам должны поддерживаться СУБД (это касается вставки/модификации кортежей в дочерней таблице, а также удаления кортежей в родительской таблице).

Ссылочная целостность может нарушиться в результате операций вставки, обновления и удаления записей в таблицах. В определении ссылочной целостности участвуют две таблицы - родительская и дочерняя, для каждой из них возможны эти операции, поэтому существует шесть различных вариантов, которые могут привести либо не привести к нарушению ссылочной целостности.

Для родительского отношения:

Вставка. Возникает новое значение первичного ключа. Существование кортежей в родительском отношении, на которые нет ссылок из дочернего отношения, допустимо, операция не нарушает ссылочной целостности.

Обновление. Изменение значения первичного ключа в кортеже может привести к нарушению ссылочной целостности.

Удаление. При удалении кортежа удаляется значение первичного ключа. Если есть кортежи в дочернем отношении, ссылающиеся на ключ удаляемого кортежа, то значения внешних ключей станут некорректными. Операция может привести к нарушению ссылочной целостности.

Для дочернего отношения:

Вставка. Нельзя вставить кортеж в дочернее отношение, если для нового кортежа значение внешнего ключа некорректно. Операция может привести к нарушению ссылочной целостности.

Обновление. При обновлении кортежа в дочернем отношении можно попытаться некорректно изменить значение внешнего ключа. Операция может привести к нарушению ссылочной целостности.

Удаление. При удалении кортежа в дочернем отношении ссылочная целостность не нарушается.

Таким образом, ссылочная целостность в принципе может быть нарушена при выполнении одной из четырех операций:

Обновление кортежей в родительском отношении.

Удаление кортежей в родительском отношении.

Вставка кортежей в дочернем отношении.

Обновление кортежей в дочернем отношении.

Основные стратегии поддержания ссылочной целостности.

Существуют две основные стратегии поддержания ссылочной целостности.

RESTRICT (ОГРАНИЧИТЬ) - не разрешать выполнение операции, приводящей к нарушению ссылочной целостности.

CASCADE (КАСКАДНОЕ ИЗМЕНЕНИЕ) - разрешить выполнение требуемой операции, но внести при этом необходимые изменения в связанных отношениях так, чтобы не допустить нарушения ссылочной целостности и сохранить все имеющиеся связи. Изменение начинается в родительском отношении и каскадно выполняется в дочерних отношениях. В реализации этой стратегии имеется одна тонкость, заключающаяся в том, что дочерние отношения сами могут быть родительскими для некоторых третьих отношений. При этом может дополнительно потребоваться выполнение какой-либо стратегии и для этой связи и т.д. Если при этом какая-

либо из каскадных операций (любого уровня) не может быть выполнена, то необходимо отказаться от первоначальной операции и вернуть базу данных в исходное состояние. Это сложная стратегия, но она не нарушает связей между родительскими и дочерними отношениями.

Эти стратегии являются стандартными и присутствуют во всех СУБД, в которых имеется поддержка ссылочной целостности.

Дополнительные стратегии поддержания ссылочной целостности.

IGNORE (ИГНОРИРОВАТЬ) - разрешить выполнять операцию без проверки ссылочной целостности. В этом случае в дочернем отношении могут появляться некорректные значения внешних ключей, вся ответственность за целостность базы данных ложится на программиста или пользователя.

SET NULL (ЗАДАТЬ ЗНАЧЕНИЕ NULL) - разрешить выполнение требуемой операции, но все возникающие некорректные значения внешних ключей изменять на null-значения. Эта стратегия имеет два недостатка. Во-первых, для нее требуется разрешение на использование null-значений. Во-вторых, кортежи дочернего отношения теряют связь с кортежами родительского отношения. Установить, с каким кортежем родительского отношения были связаны измененные кортежи дочернего отношения, после выполнения операции уже нельзя.

SET DEFAULT (ЗАДАТЬ ЗНАЧЕНИЕ ПО УМОЛЧАНИЮ) - разрешить выполнение требуемой операции, но все возникающие некорректные значения внешних ключей изменять на некоторое значение, принятое по умолчанию. Достоинство этой стратегии по сравнению с предыдущей в том, что она позволяет не пользоваться null-значениями. Установить, с какими кортежами родительского отношения были связаны измененные кортежи дочернего отношения, после выполнения такой операции тоже нельзя.

К манипуляционной составляющей относятся реляционная алгебра и реляционное исчисление (которые равносильны по возможностям).

Реляционная алгебра Кодда: 8 операций.

4 теоретико-множественные (объединение, пересечение, разность, декартово произведение отношений).

4 специальные реляционные операции (ограничение, проекция, соединение, деление отношений).

Существуют и другие алгебры.

Классификация баз данных

По технологии обработки данных базы данных подразделяются на централизованные и распределенные.

Централизованная база данных хранится в памяти одной вычислительной системы.

Распределенная база данных состоит из нескольких, возможно, пересекающихся или даже дублирующих друг друга частей, которые хранятся в различных ЭВМ вычислительной сети. Работа с такой базой осуществляется с помощью системы управления распределенной базой данных.

По способу доступа к данным базы данных разделяются на базы данных с локальным доступом и базы данных с сетевым доступом.

Архитектуры обработки данных

Сервер определим как логический процесс, который обеспечивает обслуживание запросов других процессов. Сервер не посылает результатов запрашивающему процессу до тех пор, пока не придет запрос на обслуживание. После инициирования запроса управление синхронизацией обслуживания и связей становится функцией самого сервера. В данном изложении термин «сервер» используется в смысле логического процесса, а не узла вычислительной сети, выполняющего определенные функции. В дальнейшем рассматриваются серверы баз данных.

Сервер баз данных – это логический процесс, отвечающий за обработку запросов к базам данных.

Клиента определим как процесс, посылающий серверу запрос на обслуживание. В данном контексте мы абстрагируемся от технических средств, на которых реализован процесс клиента. Главной особенностью, отличающей клиента от сервера, является то, что клиент может начать взаимодействие с сервером, а сервер, как правило, никогда не начинает подобных действий.

Так как система в целом может быть четко разделена на две части (сервер и клиенты), появляется возможность организовать работу этих двух частей на разных компьютерах. Иначе говоря, существует возможность организации распределенной обработки. Распределенная обработка предполагает, что отдельные компьютеры можно соединить коммуникационной сетью так, чтобы выполнение одной задачи обработки данных можно было распределить на несколько компьютеров этой сети.

Функциями клиента являются инициирование установления связи, запрос конкретного вида обслуживания, получение от сервера результатов и подтверждение окончания обслуживания. Хотя клиент может запросить синхронное или асинхронное уведомление об окончании обслуживания, он сам не управляет синхронизацией обслуживания и связи.

Централизованные базы данных с сетевым доступом могут иметь следующую архитектуру:

файл-сервер;

клиент-сервер;

трехуровневая архитектура («тонкий клиент» - сервер приложений - сервер базы данных).

Файл-сервер

Рис. Схема работы с БД в локальной сети с выделенным файловым сервером

Архитектура систем БД с сетевым доступом предполагает выделение одной из машин сети в качестве центральной (файловый сервер). На этот компьютер устанавливается операционная система (ОС) для выделенного сервера. На нем же хранится совместно используемая централизованная БД в виде одного или группы файлов. Все другие компьютеры сети выполняют функции рабочих станций. Файлы базы данных в соответствии с пользовательскими запросами передаются на рабочие станции, где и производится обработка информации. При большой интенсивности доступа к одним и тем же данным производительность информационной системы падает. Пользователи могут создавать также локальные БД на рабочих станциях.

Клиент-сервер

Рис. Схема работы с БД в рамках архитектуры клиент-сервер

В этой архитектуре на выделенном сервере, работающем под управлением серверной операционной системы, устанавливается специальное программное обеспечение - сервер баз данных.

Основным назначением архитектуры «клиент-сервер» является обеспечение прикладным программам клиента доступа к данным, которыми управляет сервер баз данных. Архитектура клиент-сервер позволяет нескольким клиентам совместно эффективно использовать один сервер.

СУБД подразделяется на две части: клиентскую и серверную. Основа работы сервера БД - использование языка запросов (SQL). Запрос на языке SQL, передаваемый клиентом (рабочей станцией) серверу БД, порождает поиск и извлечение данных на сервере. Извлеченные данные транспортируются по сети от сервера к клиенту. Отметим, что после обработки запроса клиента сервер баз данных посылает клиенту обратно только данные, которые удовлетворяют запросу. Тем самым, количество передаваемой по сети информации уменьшается во много раз.

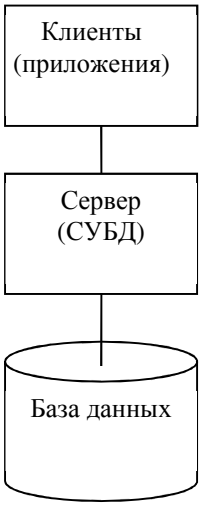


Рис. Схематическое изображение архитектуры клиент-сервер

В этом состоит ключевое отличие рассматриваемого метода от метода обработки запросов, характерного для персональных СУБД, установленных в сети (файл-сервер). В случае ахритекутуры файл-сервер при получении запроса файл, в котором находятся данные, полностью пересылается по сети в компьютер клиента, где и происходит отбор данных по критерию пользователя. В большинстве запросов отбирается менее 1% данных, а трафик в сети оказывается неоправданно высоким. Если пользователей несколько, то сеть быстро приходит в состояние перегрузки.

Преимущества и недостатки архитектуры клиент-сервер.

Преимущества:

- появляется возможность организации параллельной обработки. Для решения общей задачи применяются сразу несколько компьютеров, поэтому работа сервера (базы

данных) и клиента (приложения) осуществляется параллельно. В результате улучшаются такие показатели, как скорость обработки запросов в системе и производительность системы;

- к одному и тому же серверному компьютеру могут иметь доступ несколько разных клиентских компьютеров (что чаще всего и происходит). Поэтому одна база данных может совместно использоваться несколькими различными клиентскими системами;
- отдельный клиентский компьютер может иметь доступ к нескольким серверным компьютерам. Это весьма удобная возможность, поскольку предприятие обычно выполняет обработку данных таким образом, что полный набор всех данных не сохраняется на одном компьютере, а распределяется по нескольким разным компьютерам, причем в приложениях иногда необходим доступ к данным нескольких компьютеров;
- каждый компьютер в системе можно выбирать так, чтобы он лучше всего подходил к требованиям каждого компонента. Например, для сервера можно выбрать мощный многопроцессорный компьютер с большим количеством ОЗУ;
- такая система обладает хорошей адаптируемостью и гибкостью (масштабируемостью) – в силу модульности легко можно заменить переставший удовлетворять требованиям компонент или даже целиком какую-либо составляющую. Также можно добавить новые рабочие станции.

Однако имеются и недостатки:

- частые обращения клиента к серверу снижают производительность работы сети;
- приходится решать вопросы безопасной многопользовательской работы с данными, т.к. приложения и данные распределены между различными клиентами;
- распределенный характер построения системы обусловливает сложность ее настройки и сопровождения. В очень сложную проблему может превратиться процесс смены клиентского программного обеспечения.

Изложенные выше недостатки архитектуры клиент-сервер стимулировании поиск новых архитектур обработки данных, одним из результатов которого стала многозвенная архитектура, свободная от некоторых недостатков своей предшественницы.

Трехуровневая архитектура

Была предложена идея распределения нагрузки между тремя и более различными компьютерами, - трехуровневая архитектура. Подобная архитектура функционирует в Интранет - и Интернет - сетях. Клиент по-прежнему выполняет функции предоставления интерфейса пользователя и, возможно, некоторые не очень сложные и ресурсоемкие операции обработки данных, другие этапы функционирования системы теперь распределены между несколькими компьютерами – серверами баз данных и серверами приложений. Серверы баз данных управляют данными, а серверы приложений выполняют все вычисления, связанные с реализацией бизнес-логики.

Дадим ряд определений, существенных для описания многозвенной архитектуры.

В ее состав входит тонкий клиент – определим его как процесс, посылающий запрос на обслуживание и способный осуществить отображение его результатов на основе некоторого универсального протокола выдачи информации. Основное отличие тонких клиентов от обычных – способность предоставления пользователю интерфейса для решения любых задач, низкая стоимость внедрения, администрирования и поддержки. Как правило, это браузер, программа просмотра сценариев на каком-либо языке разметки. Браузер может поддерживать с помощью run-time расширений и другие форматы файлов.

Сервер баз данных – это логический процесс, отвечающий за обработку запросов к базе данных.

Сервер приложений – совокупность логических процессов, реализующих бизнес-логику на основании данных, предоставляемых сервером баз данных и передающих результаты

вычислений универсальному клиенту через некоторую среду передачи данных. Администрирование и обслуживание приложений осуществляется полностью на сервере приложений, а не на стороне клиента, поэтому обновлять программные модули универсального клиента приходится довольно редко.

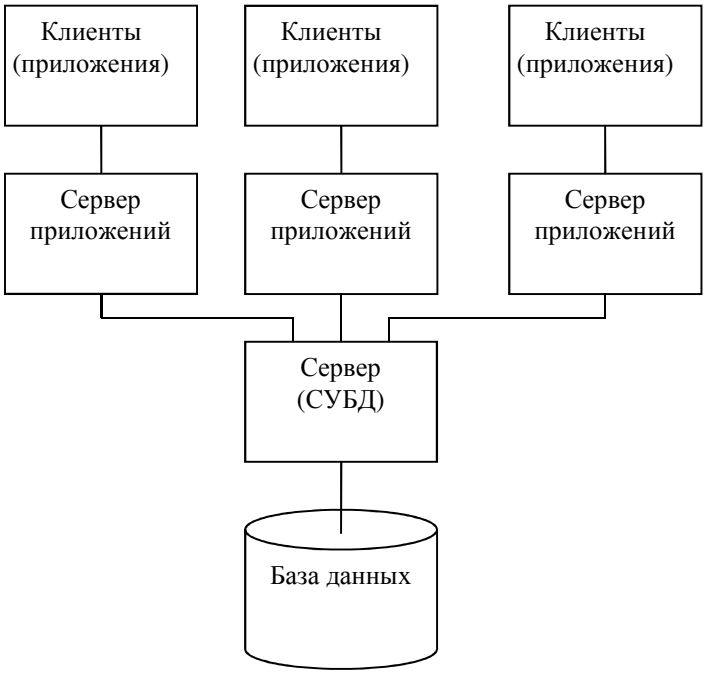


Рис. Схематическое изображение трехуровневой архитектуры

Основными экономическими преимуществами данной архитектуры являются:

- относительно низкие затраты на внедрение и эксплуатацию;
- высокая способность к интеграции существующих информационных ресурсов;
- прикладные программные средства доступны с любого клиентского рабочего места;
- минимальный состав программно-технических средств на клиентском рабочем месте.

Архитектура СУБД Oracle

Описание СУБД Oracle

СУБД Oracle является не только одной из первых СУБД, но и лидирует на данный момент среди систем на платформах Unix и Windows.

Некоторые преимущества СУБД Oracle:

- высочайшая надежность;
- поддержка многих компьютерных платформ;
- поддержка всевозможных вариантов архитектур (многопроцессорных кластеров, систем с массовым параллелизмом и т.д.);
- идентичность кода различных версий сервера б.д. для всех платформ, гарантирующая предсказуемость работы Oracle на разных платформах;
- возможности разбиения крупных б.д. на разделы;
- широкие возможности средств защиты информации;
- эффективные методы повышения скорости обработки запросов;
- распараллеливание операций в запросе;
- широкий спектр средств разработки, мониторинга и администрирования;
- ориентация на Интернет-технологии;
- поддержка XML в хранимых процедурах;
- поддержка OLAP (Online Analytical Processing) и т.д.

Опираясь на концепцию многозвенной архитектуры, Oracle предлагает три базовых элемента информационной системы:

- сервер баз данных Oracle;
- универсальный сервер приложений Oracle Application Server;
- набор драйверов в стандарте JDBC, специально оптимизированных для доступа из Java-программ к Oracle, а также SQLJ – поддержка операторов SQL, встроенных в программы Java.

Этот набор не является жестко заданным. При проектировании конкретной системы следует учитывать все особенности ее функционирования и, например, использование web-сервера Apache и языка Perl может оказаться эффективнее применения web-расширений языка PL/SQL для Oracle Application Server. Кроме того, в интерпретации Oracle многозвенная архитектура имеет еще одну особенность – она является расширяемой.

Архитектура сервера Oracle

Сервер баз данных – основа любой распределенной системы. Доминирование технологий Oracle в первую очередь вызвано удачными решениями при проектировании своего основного продукта.

Процессы

В этом разделе «процесс» будет использоваться как синоним «потока» в операционных системах, где сервер Oracle реализован с помощью потоков системного процесса. Так, например, если описывается процесс DBWn, в среде Windows ему соответствует поток DBWn, а в ОС UNIX - процесс.

В экземпляре Oracle есть три класса процессов.

- Серверные процессы. Они выполняют запросы клиентов.
- Фоновые процессы. Это процессы, которые начинают выполняться при запуске экземпляра и решают различные задачи поддержки базы данных, такие как запись блоков на диск, поддержка активного журнала повторного выполнения, удаление прекративших работу процессов и т.д.
- Подчиненные процессы. Они подобны фоновым процессам, но выполняют, корме того, действия от имени фонового или серверного процесса.

Серверные процессы

Выделенные и разделяемые серверы решают одну и ту же задачу: обрабатывают передаваемые им SQL-операторы. При получении запроса SELECT * FROM EMP именно выделенный/разделяемый сервер Oracle будет разбирать его и помещать в разделяемый пул (или находить соответствующий запрос в разделяемом пуле). Именно этот процесс создает план выполнения запроса. Этот процесс реализует план запроса, находя необходимые данные в буферном кэше или считывая данные в буферный кэш с диска.

С клиентским приложением скомпонованы библиотеки Oracle. Они обеспечивают программный интерфейс (Application Program Interface — API) для взаимодействия с базой данных. Функции API "знают", как передавать запрос к базе данных и обрабатывать возвращаемый курсор. Они обеспечивают преобразование запросов пользователя в передаваемые по сети пакеты, обрабатываемые выделенным сервером. Эти функции обеспечивает компонент Net8 — сетевое программное обеспечение/протокол, используемое Oracle для клиент/серверной обработки. Сервер Oracle использует такую архитектуру, даже если протокол Net8 не нужен (когда клиент и сервер работают на одной и той же машине).

Сервер Oracle может работать в двух режимах: выделенного и разделяемого сервера (MTS).

Режим выделенного сервера

Режим выделенного сервера — наиболее широко используемый способ подключения к СУБД Oracle для всех приложений, использующих SQL-запросы. Режим выделенного сервера проще настроить и он обеспечивает самый простой способ подключения. При этом требуется минимальное конфигурирование.

В режиме выделенного сервера имеется однозначное соответствие между клиентскими сеансами и серверными процессами (или потоками). Если имеется 100 сеансов на UNIX-машине, будет 100 процессов, работающих от их имени.

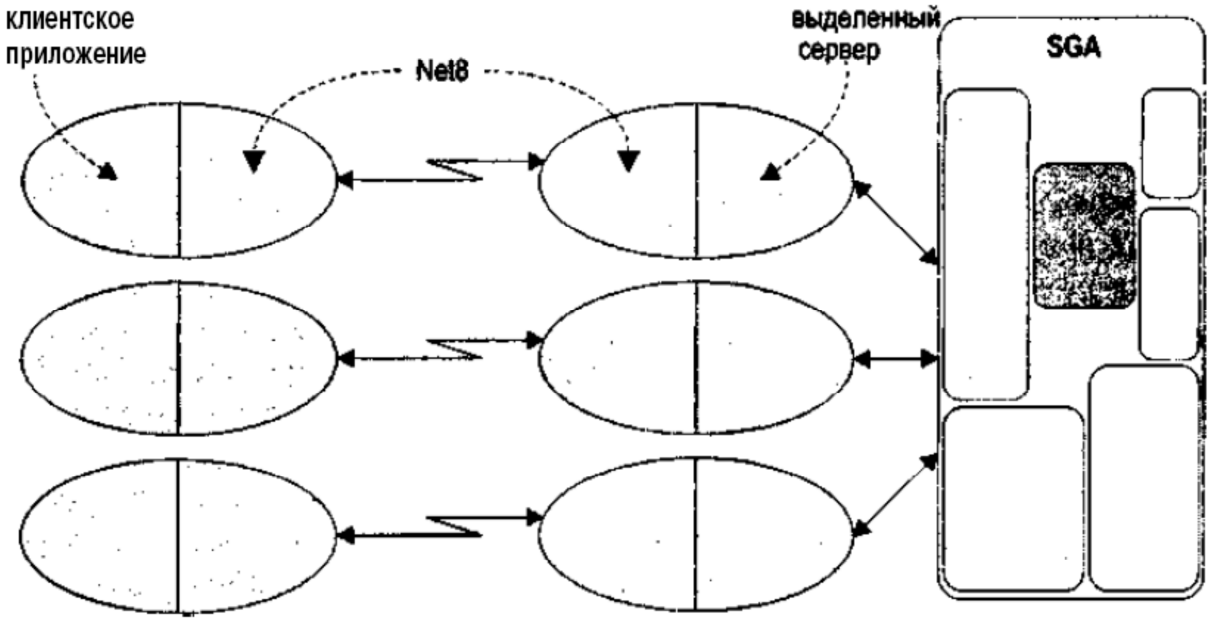


Рис. Схематическое изображение работы СУБД Oracle в режиме выделенного сервера

Эти серверные процессы "порождаются", или создаются, процессом прослушивания Oracle Net8 Listener. Если процесс прослушивания не задействован, механизм во многом аналогичен, но вместо создания выделенного сервера процессом прослушивания, процесс создается непосредственно клиентским процессом.

Режим MTS

Теперь давайте более детально рассмотрим другой тип серверных процессов — разделяемый серверный процесс. Клиентское приложение подключается к процессу прослушивания Net8 и перенаправляется на процесс-диспетчер. Диспетчер играет роль канала передачи информации между клиентским приложением и разделяемым серверным процессом. Ниже представлена схема подключения к базе данных через разделяемый сервер:

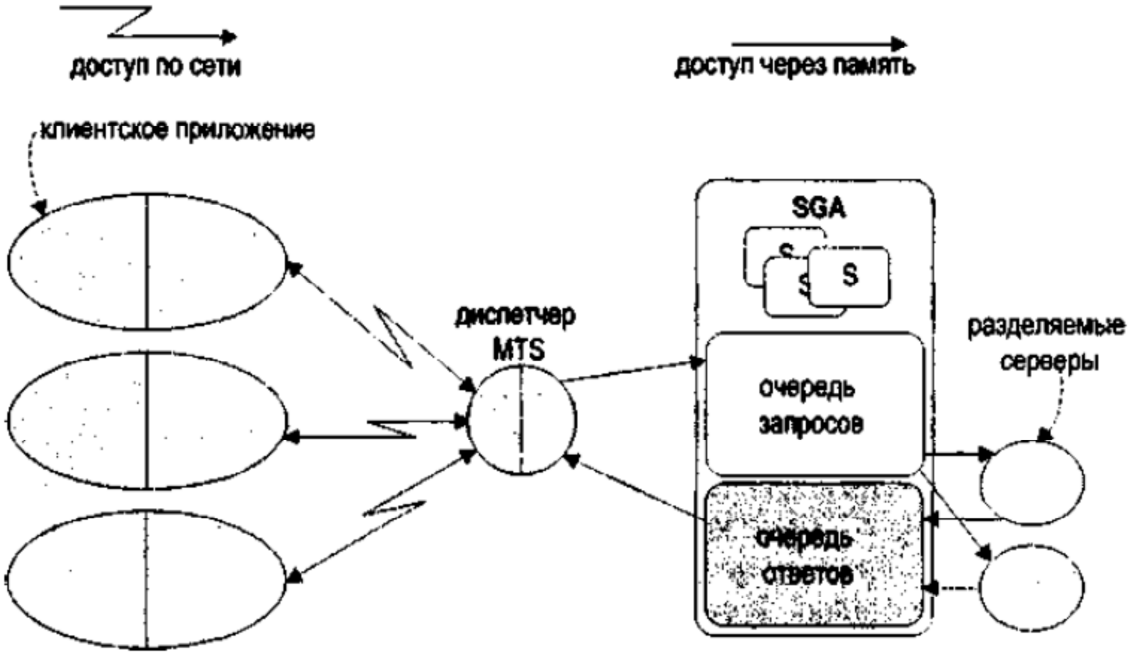


Рис. Схематическое изображение работы СУБД Oracle в режиме MTS

Диспетчеров MTS для любого экземпляра можно сгенерировать несколько, но часто для сотен и даже тысяч пользователей используется один диспетчер. Диспетчер отвечает за получение входящих запросов от клиентских приложений и их размещение в очереди запросов в области SGA. Первый свободный разделяемый серверный процесс, по сути, ничем не отличающийся от выделенного серверного процесса, выберет запрос из очереди и подключится к области UGA соответствующего сеанса (прямоугольника с 'S' на представленной выше схеме). Разделяемый сервер обработает запрос и поместит полученный при его выполнении результат в очередь ответов. Диспетчер постоянно следит за появлением результатов в очереди и передает их клиентскому приложению. С точки зрения клиента нет никакой разницы между подключением к выделенному серверу и подключением в режиме MTS, — они работают одинаково. Различие возникает только на уровне экземпляра.

Сравнение режима выделенного сервера и режима MTS.

В режиме MTS соответствие — многие к одному (много клиентов и один разделяемый сервер). Как следует из названия, разделяемый сервер — общий ресурс, а выделенный — нет. При использовании общего ресурса необходимо стараться не монополизировать его надолго. Правило номер один для режима MTS: убедитесь, что транзакции выполняются быстро. Они могут выполняться часто, но должны быть короткими. В противном случае будут наблюдаться все признаки замедления работы системы из-за монополизации общих ресурсов несколькими процессами. В экстремальных случаях, если все разделяемые серверы заняты, система "зависает".

Режим MTS не подходит для хранилища данных. В такой системе выполняются запросы продолжительностью одна, две, пять и более минут. Для режима MTS это "смертельно".

Итак, какие же преимущества дает режим MTS, если учитывать, для какого типа транзакций он предназначен? Режим MTS позволяет добиться следующего:

- сократить количество процессов/потоков операционной системы. В системе с тысячами пользователей ОС может быстро оказаться перегруженной при попытке управлять тысячами процессов. В обычной системе одновременно активна лишь небольшая часть этих тысяч пользователей;
- искусственно ограничить степень параллелизма.

Рис. Иллюстрация зависимости выполняемых транзакций в секунду от количества одновременно работающих пользователей

Сначала при добавлении одновременно работающих пользователей количество транзакций растет. С какого-то момента, однако, добавление новых пользователей не увеличивает количества выполняемых в секунду транзакций: оно стабилизируется. Пропускная способность достигла максимума, и время ожидания ответа начинает расти (каждую секунду выполняется то же количество транзакций, но пользователи получают результаты со все возрастающей задержкой. При дальнейшем добавлении пользователей пропускная способность начинает падать. Количество одновременно работающих пользователей перед началом этого падения и является максимально допустимой степенью параллелизма в системе. Дальше система переполняется запросами, и образуются очереди. С этого момента система не справляется с нагрузкой. Не только существенно увеличивается время ответа, но и начинает падать пропускная способность системы. Если ограничить количество одновременно работающих пользователей до числа, непосредственно предшествующего падению, можно обеспечить максимальную пропускную способность и приемлемое время ответа для большинства пользователей. Режим MTS позволяет ограничить максимальную степень параллелизма в системе до этого количества одновременно работающих пользователей.

- сократить объем памяти, необходимый системе. Это одна из наиболее часто упоминаемых причин использования режима MTS: сокращается объем памяти, необходимой для поддержки определенного количества пользователей. Да, сокращается, но не настолько, как можно было бы ожидать. Помните, что при использовании режима MTS область UGA помещается в SGA. Это означает, что при переходе на режим MTS необходимо точно оценить суммарный объем областей UGA и выделить место в области SGA с помощью параметра инициализации LARGE_POOL. Поэтому размер области SGA при использовании режима MTS обычно очень большой. Эта память выделяется заранее и поэтому может использоваться только СУБД. Экономия связана с уменьшением количества выделяемых областей PGA. Каждый выделенный/разделяемый сервер имеет область PGA. В ней хранится информация процесса. В ней располагаются области сортировки, области хешей и другие структуры процесса. Именно этой памяти для системы надо меньше, если используется режим MTS. При переходе с 5000 выделенных серверов на 100 разделяемых освобождается 4900 областей PGA — именно такой объем памяти и экономится в режиме MTS.

Фоновые процессы

Экземпляр Oracle состоит из двух частей: области SGA и набора фоновых процессов. Фоновые процессы выполняют рутинные задачи сопровождения, обеспечивающие работу СУБД. Каждый из этих процессов решает конкретную задачу, но работает в координации с остальными. Например, когда процесс, записывающий файлы журнала, заполняет один журнал и переходит на следующий, он уведомляет процесс, отвечающий за архивирование заполненного журнала, что для него есть работа.

Есть два класса фоновых процессов: предназначенные исключительно для решения конкретных задач и решающие множество различных задач.

Фоновые процессы, предназначенные для решения конкретных задач:

PMON. Монитор процессов, Этот процесс отвечает за очистку после нештатного прекращения подключений. Процесс PMON следит за всеми процессами Oracle и либо перезапускает их, либо прекращает работу экземпляра, в зависимости от ситуации. Еще одна функция процесса PMON в экземпляре — регистрировать экземпляр в процессе прослушивания протокола Net8.

SMON. Монитор системы. SMON — это процесс, занимающийся всем тем, от чего "отказываются" остальные процессы. Это своего рода "сборщик мусора" для базы данных.

RECO. Восстановление распределенной базы данных. Процесс RECO восстанавливает транзакции, оставшиеся в готовом состоянии из-за сбоя или потери связи в ходе двухэтапной фиксации (2PC).

CKPT. Обработка контрольной точки. Процесс обработки контрольной точки содействует обработке контрольной точки, обновляя заголовки файлов данных.

DBWn. Запись блоков базы данных. Процесс записи блоков базы данных — фоновый процесс, отвечающий за запись измененных блоков на диск. Процесс DBWn записывает измененные блоки из буферного кэша, чтобы освободить пространство в кэше или в ходе обработки контрольной точки.

LGWR. Запись журнала. Процесс LGWR отвечает за сброс на диск содержимого буфера журнала повторного выполнения, находящегося в области SGA.

ARCn. Архивирование. Задача процесса ARCn — копировать в другое место активный файл журнала повторного выполнения, когда он заполняется процессом LGWR.

BSP (сервер блоков), LMON (контроль блокировок), LMD (демон диспетчера блокировок), LCKn (блокирование) - процессы, использующиеся исключетильно в среде Oracle Parallel Server (OPS, конфигурация сервера Oracle, поддерживающая несколько экземпляров одной базы данных на различных машинах в кластерной среде).

Служебные фоновые процессы

Эти фоновые процессы необязательны — они запускаются в случае необходимости. Они реализуют средства, необязательные для штатного функционирования базы данных. Использование этих средств инициируется явно или косвенно, при использовании возможности, требующей их запуска.

SNPn - обработка снимков (очереди заданий), QMNn — монитор очередей, EMNn — монитор событий.

Подчиненные процессы

В сервере Oracle есть два типа подчиненных процессов — ввода/вывода (I/O slaves) и параллельных запросов (Parallel Query slaves).

Подчиненные процессы ввода/вывода используются для эмуляции асинхронного ввода/вывода в системах или на устройствах, которые его не поддерживают.

Подчиненные процессы параллельных запросов используются средствами распараллеливания запросов к базе данных.

Структура памяти

Память, используемая сервером Oracle, имеет следующую структуру:

SGA, System Global Area – глобальная область системы. Это большой совместно используемый сегмент памяти, к которому обращаются все процессы Oracle;

PGA, Process Global Area – глобальная область процесса. Это приватная область памяти процесса или потока, недоступная другим процессам/потокам;

UGA, User Global Area – глобальная область пользователя. Это область памяти, связанная с сеансом.

Области PGA и UGA

Как уже было сказано, PGA — это область памяти процесса. Эта область памяти используется одним процессом или одним потоком. Она недоступна ни одному из остальных процессов/потоков в системе. Область PGA обычно выделяется с помощью библиотечного вызова malloc() языка C и со временем может расти (или уменьшаться).

Область PGA никогда не входит в состав области SGA — она всегда локально выделяется процессом или потоком.

Область памяти UGA хранит состояние сеанса, поэтому всегда должна быть ему доступна. Местонахождение области UGA зависит исключительно от конфигурации сервера Oracle. Если сконфигурирован режим MTS, область UGA должна находиться в структуре памяти, доступной всем процессам, следовательно, в SGA. В этом случае сеанс сможет использовать любой разделяемый сервер, так как каждый из них сможет прочитать и записать данные сеанса. При подключении к выделенному серверу это требование общего доступа к информации о состоянии сеанса снимается, и область UGA становится почти синонимом PGA, — именно там информация о состоянии сеанса и будет располагаться.

При работе в режиме MTS необходимо задать такой размер области SGA, чтобы в ней хватило места под области UGA для предполагаемого максимального количества одновременно подключенных к базе данных пользователей. Поэтому область SGA в экземпляре, работающем в режиме MTS, обычно намного больше, чем область SGA аналогично сконфигурированного экземпляра, работающего в режиме выделенного сервера.

Область SGA

Каждый экземпляр Oracle имеет одну большую область памяти, которая называется SGA, — глобальная область системы. Это большая разделяемая структура, к которой обращаются все процессы Oracle. Ее размер варьируется от нескольких мегабайт в небольших тестовых системах, до сотен мегабайт в системах среднего размера и множества гигабайт в больших системах.

В SGA содержатся внутренние структуры данных для кэширования данных с диска, кэширования данных повторного выполнения перед записью на диск, хранения планов выполнения разобранных операторов SQL и т.д. Например, в SGA в течение некоторого времени хранится дерево синтаксического разбора и план выполнения для каждого оператора SQL, и, если происходит повторное выполнение такого же оператора, то повторный анализ не производится и используется находящийся в SGA план выполнения. Таким образом, повышается быстродействие системы за счет устранения дублирования операций.

Область SGA разбита на несколько областей и может быть представлена следующим образом:

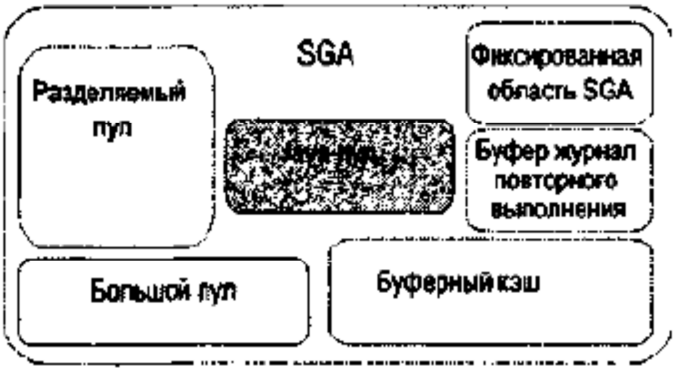


Рис. Структура памяти SGA

Фиксированная область SGA

Фиксированная область SGA — это часть области SGA, размер которой зависит от платформы и версии. Она "компилируется" в двоичный модуль сервера Oracle при установке. Фиксированная область SGA содержит переменные, которые указывают на другие части SGA, а также переменные, содержащие значения различных параметров. Размером фиксированной

области SGA управлять нельзя. Можно рассматривать эту область как "загрузочную" часть SGA, используемую сервером Oracle для поиска других компонентов SGA.

Буфер журнала повторного выполнения

Буфер журнала повторного выполнения используется для временного кэширования данных активного журнала повторного выполнения перед записью на диск. Поскольку перенос данных из памяти в память намного быстрее, чем из памяти — на диск, использование буфера журнала повторного выполнения позволяет существенно ускорить работу сервера. Данные не задерживаются в буфере журнала повторного выполнения надолго.

Буферный кэш

До сих пор мы рассматривали небольшие компоненты области SGA. Теперь переходим к составляющей, которая достигает огромных размеров. В буферном кэше сервер Oracle хранит блоки базы данных перед их записью на диск, а также после считывания с диска. Это принципиально важный компонент SGA. Если сделать его слишком маленьким, запросы будут выполняться годами. Если же он будет чрезмерно большим, пострадают другие процессы (например, выделенному серверу не хватит пространства для создания области PGA, и он просто не запустится). Интенсивно используемые блоки кэшируются надолго, а редко используемые — долго в кэше не задерживаются. Буферный кэш — первое и основное средство настройки производительности сервера. Он существует исключительно для ускорения очень медленного процесса ввода/вывода.

Разделяемый пул

Разделяемый пул — один из наиболее важных фрагментов памяти в области SGA, особенно для обеспечения производительности и масштабируемости.

В разделяемом пуле сервер Oracle кэширует различные "программные" данные. Здесь кэшируются результаты разбора запроса. Перед повторным разбором запроса сервер Oracle просматривает разделяемый пул в поисках готового результата. Выполняемый сеансом PL/SQL-код тоже кэшируется здесь, так что при следующем выполнении не придется снова читать его с диска. PL/SQL-код в разделяемом пуле не просто кэшируется, — появляется возможность его совместного использования сеансами. Если 1000 сеансов выполняют тот же код, загружается и совместно используется всеми сеансами лишь одна копия этого кода. Это же относится к таким объектам, как курсоры. Сервер Oracle хранит в разделяемом пуле параметры системы. Здесь же хранится кэш словаря данных, содержащий информацию об объектах базы данных.

В разделяемом пуле отсутствует понятие освобождения фрагмента памяти. Память выделяется, используется, а затем перестает использоваться. Через некоторое время, если эту память необходимо использовать повторно, сервер Oracle позволит изменить содержимое устаревшего фрагмента (принцип пула KEEP).

Слишком маленький разделяемый пул может снизить производительность настолько, что система будет казаться зависшей. Слишком большой разделяемый пул может привести к такому же результату. Неправильное использование разделяемого пула грозит катастрофой.

Большой пул

Большой пул назван так не потому, что это "большая" структура, а потому, что используется для выделения больших фрагментов памяти — больших, чем те, для управления которыми создавался разделяемый пул. Большой пул — это область памяти, управляемая по принципу пула RECYCLE (из пула RECYCLE блок выбрасывается сразу после использования.). После освобождения фрагмента памяти он может использоваться другими процессами. Большой пул используется сервером в режиме MTS для размещения памяти сеанса, средствами распараллеливания Parallel Execution для буферов сообщений и при резервном копировании с помощью RMAN для буферов дискового ввода/вывода;

Java-пул

Java-пул был добавлен в версии 8.1.5 для поддержки работы Java-машины JVM в базе данных. Если поместить хранимую процедуру на языке Java или компонент EJB (Enterprise

JavaBean) в базу данных, сервер Oracle будет использовать этот фрагмент памяти при обработке соответствующего кода.

Отметим, что при работе в режиме MTS UGA выделяется из Java-пула (для хранения информации о состоянии сеансов), из разделяемого пула или из большого пула, если он выделен.

Файлы

Термины "база данных" и "экземпляр" вызывают большую путаницу. В соответствии с принятой в Oracle терминологией, эти понятия определяются так:

• база данных — набор файлов с базой данных;

• экземпляр — набор процессов Oracle и область SGA.

Эти два термина иногда взаимозаменяемы, но представляют принципиально разные концепции. Взаимосвязь между ними такова, что база данных может быть смонтирована и открыта в нескольких экземплярах. Экземпляр может смонтировать и открыть только одну базу данных в каждый момент времени. Не обязательно открывать и монтировать одну и ту же базу данных при каждом запуске экземпляра. Итак, в большинстве случаев между базой данных и экземпляром имеется отношение один к одному.

Процессы в ходе своей работы используют файлы, совокупность которых является физическим представлением базы данных. Существуют три основные группы файлов, составляющих базу данных: файлы базы данных, управляющие файлы и журнальные файлы. В файлах базы данных располагаются данные, а управляющие и журнальные файлы поддерживают функциональность сервера. Все три набора файлов должны присутствовать, быть открытыми и доступными Oracle.

Файлы б.д. используют единицы выделения пространства под объекты.

Сегменты — это области на диске, выделяемые под объекты — таблицы, индексы, сегменты отката и т.д. При создании таблицы создается сегмент таблицы. При создании фрагментированной таблицы создается по сегменту для каждого фрагмента. При создании индекса создается сегмент индекса и т.д. Каждый объект, занимающий место на диске, хранится в одном сегменте. Есть сегменты отката, временные сегменты, сегменты кластеров, сегменты индексов и т.д.

Сегменты, в свою очередь, состоят из одного или нескольких экстентов. Экстент — это непрерывный фрагмент пространства в файле. Каждый сегмент первоначально состоит хотя бы из одного экстента, причем для некоторых объектов требуется минимум два экстента (в качестве примера можно назвать сегменты отката). Чтобы объект мог вырасти за пределы исходного экстента, ему необходимо выделить следующий экстент. Этот экстент не обязательно должен выделяться рядом с первым; он может находиться достаточно далеко от первого, но в пределах экстента в файле пространство всегда непрерывно. Размер экстента варьируется от одного блока до 2 Гбайт.

Экстенты состоят из блоков. Блок — наименьшая единица выделения пространства в Oracle. В блоках и хранятся строки данных, индексов или промежуточные результаты сортировок. Именно блоками сервер Oracle обычно выполняет чтение и запись на диск. Блоки в Oracle бывают чаще всего размером 2 Кбайт, 4 Кбайт или 8 Кбайт (хотя допустимы также блоки размером 16 Кбайт и 32 Кбайт).

Записи заносятся в таблицу в физической последовательности. Первая запись хранится в первом блоке первого экстента. Следующие за ней записи заносятся в первый блок до тех пор, пока он не заполнится. Блок считается заполненным, когда очередной записи не хватает оставшегося свободного пространства в блоке. Когда по мере занесения записей заполняются все блоки первого экстента, выделенного таблице, для данных выделяется первый дополнительный экстент. Когда будет заполнен и этот экстент, сервер выделит следующий.

После выделения таблице экстента данных или индекса он не освобождается до тех пор, пока таблица не будет уничтожена. Однако если все данные удалены из блока данных, то этот блок становится доступным для повторного использования этой же таблицей или кластером.

Табличное пространство — это контейнер с сегментами. Каждый сегмент принадлежит к одному табличному пространству. В табличном пространстве может быть много сегментов. Все экстенты сегмента находятся в табличном пространстве, где создан сегмент. Сегменты никогда не переходят границ табличного пространства. С табличным пространством, в свою очередь, связан один или несколько файлов данных. Экстент любого сегмента табличного пространства целиком помещается в одном файле данных. Однако экстенты сегмента могут находиться в нескольких различных файлах данных.

Файлы журнала повторного выполнения принципиально важны для базы данных Oracle. Это журналы транзакций базы данных. Они используются только для восстановления при сбое экземпляра или носителя или при поддержке резервной базы данных на случай сбоев.

Практически каждое действие, выполняемое в СУБД Oracle, генерирует определенные данные повторного выполнения, которую надо записать в активные файлы журнала повторного выполнения. При вставке строки в таблицу конечный результат этой операции записывается в журналы повторного выполнения. При удалении строки записывается факт удаления. При удалении таблицы в журнал повторного выполнения записываются последствия этого удаления. Есть два типа файлов журнала повторного выполнения: активные и архивные.

В каждой базе данных Oracle есть как минимум два активных файла журнала повторного выполнения. Эти активные файлы журнала повторного выполнения имеют фиксированный размер и используются циклически. Сервер Oracle выполняет запись в файл журнала 1, а когда доходит до конца этого файла, — переключается на файл журнала 2 и переписывает его содержимое от начала до конца.

Чтобы понять, как используются активные журналы повторного выполнения, надо разобраться с обработкой контрольной точки, использованием буферного кэша базы данных и рассмотреть функции процесса записи блоков базы данных (Database Block Writer — DBWn).

В буферном кэше базы данных временно хранятся блоки базы данных. При чтении блоки запоминаются в этом кэше (предполагается, что в дальнейшем их не придется читать с диска). При изменении блока путем обновления одной из его строк изменения выполняются в памяти, в блоках буферного кэша. Информация, достаточная для повторного выполнения этого изменения, записывается в буфер журнала повторного выполнения - еще одну структуру данных в области SGA. При фиксации изменений с помощью оператора COMMIT сервер Oracle не записывает на диск все измененные блоки в области SGA. Он только записывает в активные журналы повторного выполнения содержимое буфера журнала повторного выполнения. Пока измененный блок находится в кэше, а не на диске, содержимое соответствующего активного журнала может быть использовано в случае сбоя экземпляра. Если измененный блок находится в кэше и не записан на диск, мы не можем повторно записывать соответствующий файл журнала повторного выполнения.

Фоновый процесс DBWn сервера Oracle отвечает за освобождение буферного кэша при заполнении и обработку контрольных точек. Обработка контрольной точки состоит в сбросе измененных блоков из буферного кэша на диск. Сервер Oracle делает это автоматически, в фоновом режиме.

База данных Oracle может работать в двух режимах — NOARCHIVELOG и ARCHIVELOG. Эти режимы отличаются тем, что происходит с файлом журнала повторного выполнения до того как сервер Oracle его перепишет. В режиме ARCHIVELOG сохраняется копия данных активного журнала повторного выполнения в архивный журнал повторного выполнения. В режиме NOARCHIVELOG копия не сохраняется активный журнал переписывается. В последнем случае теряется возможность восстановления данных с резервной копии до текущего момента.

Если база данных не работает в режиме ARCHIVELOG, данные рано или поздно будут потеряны. Работать в режиме NOARCHIVELOG можно только в среде разработки или тестирования.

Журнал сообщений — это файл на сервере, содержащий информационные сообщения сервера, например, о запуске и останове, а также уведомления об исключительных ситуациях, вроде незавершенной обработки контрольной точки.

Информация о структуре объектов базы данных, их расположении, правах доступа и т.п. хранится в словаре данных. Словарь данных – база метаданных о базе данных. Информация словаря данных хранится в виде таблиц, над которыми созданы многочисленные представления, и пользователь, обладающий необходимыми правами доступа, может получить необходимую информацию по текущему состоянию базы, используя запросы на языке SQL.

Все представления словаря данных можно разделить на три группы: DBA-представления, содержащие информацию обо всех объектах базы данных; ALL-представления, содержащие информацию только о тех объектах, которые доступны пользователю; USER-представления, содержащие информацию обо всех объектах базы данных, принадлежащих пользователю. Например, сведения о таблицах, которые находятся в схеме текущего пользователя, можно посмотреть в представлении USER_TABLES.

Конфигурирование

Дадим общие сведения, поясняющие процесс конфигурирования.

Запущенный экземпляр получает уникальный идентификатор – SID. После запуска администратором экземпляра базы данных и ее открытия пользователи могут присоединяться к базе данных, но при этом пользователь должен знать соответствующий идентификатор экземпляра. В одном и том же базовом каталоге ORACLE_HOME может быть несколько баз данных, так что необходимо иметь возможность уникально идентифицировать их и соответствующие конфигурационные файлы. Обычно информация об идентификаторе экземпляра описывается в файле конфигурации сервисов Net и пользователь использует ее косвенно через указание строки связи.

Каждый сервер обладает именем. Имя сервера относится к физической машине, на которой находится база данных Oracle. В сети каждый сервер имеет свой конкретный, уникальный сетевой адрес. Обратиться к серверу можно по имени или по сетевому адресу. Для идентификации базы данных на сервере служит имя экземпляра б.д., включающее название базы данных и домен.

Чтобы взаимодействие с сервером Oracle было возможным, на сетевом узле, где находится сервер баз данных, должен быть запущен как минимум один прослушивающий процесс. После его запуска появляется возможность приема входящих запросов и их обработки.

При запуске прослушивающий процесс считывает необходимую ему управляющую информацию из специального файла параметров. Место расположения этого файла зависит от операционной системы, его имя, как правило, listener.ora. Чаще всего этот файл расположен в каталоге %ORACLE_HOME%\Network\Admin.

С помощью одного файла можно настроить более одного прослушивающего процесса, поэтому секция настройки конкретного процесса начинается с указания его имени.

```
LISTENER =
  (ADDRESS_LIST =
    (ADDRESS = (PROTOCOL = IPC)(KEY = EXTPROC_FOR_XE))
    (ADDRESS = (PROTOCOL = TCP)(HOST = toshiba)(PORT = 1521))
  )

SID_LIST_LISTENER =
  (SID_LIST =
    (SID_DESC =
      (SID_NAME = PLSExtProc)
      (ORACLE_HOME = G:\oraclexe\app\oracle\product\10.2.0\server)
      (PROGRAM = extproc)
    )
```

```
  (SID_DESC =
    (SID_NAME = CLRExtProc)
    (ORACLE_HOME = G:\oraclexe\app\oracle\product\10.2.0\server)
    (PROGRAM = extproc)
  )
 )
```

В нашем примере – это LISTENER. В секции ADDRESS указывается нахождение сервера базы данных в сети и используемого для связи с ним протокола. Символическое имя сервера (toshiba) должно быть правильно отображено на реальный IP-адрес. Здесь можно задать и IP-адрес.

Во втором разделе (SID_LIST_LISTENER) указывается список экземпляров, баз данных и сервисов, с которыми взаимодействует прослушивающий процесс. Парметр SID_NAME содержит системный идентификатор базы данных (SID) (PLSExtProc, CLRExtProc). Его значение указывается при создании базы данных.

Остальные разделы необязательны и предназначены для тонкой настройки прослушивающего процесса.

Для запуска прослушивающего процесса предназначена утилита LSNRCTL. В Windows достаточно запустить соответствующую службу.

Для каждой базы данных существует ее файл конфигурации, который создается при создании базы данных. Файл параметров инициализации экземпляра обычно называют файлом init или файлом init.ora. Это название происходит от стандартного имени этого файла, — init<ORACLE_SID>.ora. Например, экземпляр со значением SID, равным tkyte816, обычно имеет файл инициализации inittkyte816.ora. Без файла параметров инициализации нельзя запустить экземпляр Oracle.

Файл init<ORACLE_SID>.ora расоплагается в %ORACLE_HOME\database.

В Oracle файл init<ORACLE_SID>.ora имеет очень простую конструкцию. Он представляет собой набор пар имя параметра/значение. Файл init.ora может иметь такой вид:

```
db_name = "testDatabase"
db_block_size = 8192
control_files = ("C:\oradata\control01.ctl", "C:\oradata\control02.ctl")
```

Из файла параметров инициализации (init.ora) экземпляр может узнать, где находятся управляющие файлы, а в управляющем файле описано местонахождение файлов данных и файлов журнала повторного выполнения. В управляющих файлах хранится и другие необходимые серверу Oracle сведения, в частности время обработки контрольных точек, имя базы данных, дата и время создания базы данных, хронология архивирования журналов повторного выполнения и т.д.

Сетевые средства Oracle обеспечивают поддержку трех верхних уровней модели OSI – сеансового, представления и прикладного. Совокупность сетевых средств Oracle называется Net сервисами Oracle. Net сервисы Oracle организуют взаимодействие приложений с сетевым программным обеспечением. Уровень представления служит интерфейсом между различными приложениями и программным обеспечением сеансового уровня. Например, сюда относятся драйверы Microsoft ODBC, которые позволяют организовать взаимодействие приложений с Net сервисами Oracle. Некоторые приложения могут обращаться напрямую к уровню представления. К ним относится большинство утили Oracle, а для разработки собственных приложений целесообразно использовать стандартные механизмы.

Для настройки Net сервисов Oracle на рабочих станциях предназначен конфигурационный файл tnsnames.ora. В нем находится информация о серверах и способах связи с ними. Список параметров способа связи зависит от используемого протокола.

Пример файла tnsnames.ora:

```
Orabase.World =
  (DESCRIPTION =
    (ADDRESS = (PROTOCOL = TCP)(HOST = toshiba)(PORT = 1521))
```

```
    (CONNECT_DATA =
      (SERVER = DEDICATED)
      (SERVICE_NAME = XE)
      (SID = PLSExtProc)
    )
  )
```

Для настройки соединений также служит файл sqlnet.ora. Он содержит некоторые более тонкие параметры конфигурации Net сервисов Oracle. Для большинства простых систем этот файл не требуется изменять.

Для соединения с базой данных клиентская часть должна разрешить строку соединения, то есть фактически установить сетевой маршрут к сервису, который включает нахождение прослушивающего процесса через протокольный адрес и глобальное имя базы данных, содержащее как имя базы данных, так и имя ее домена. Клиентская часть раскладывает глобальное имя базы данных на составляющие, чтобы определить, с какой базой данных необходимо связаться. После определения нахождения сервера и имени базы данных клиентская программа обращается к прослушивающему процессу, запущенному на этом сервере, чтобы установить соединение с базой данных.

Когда клиент выполняет какие-либо действия, прослушивающий процесс проверяет переданную ему информацию, сверяет данные, зарегистрированные в сервисе базы данных со своим файлом listener.ora, и выясняет, можно ли выполнить запрос пользователя на соединение. Если все корректно и соединение можно установить, то для обслуживания клиента прослушивающий процесс создает новый пользовательский процесс или использует процесс, созданный заранее. Этот процесс делает возможной связь между пользователем и базой данных. После того как данный клиентский запрос обработан, процесс продолжает прослушивать сеть и принимать новые запросы на соединение.

Администрирование

Для просмотра текущей активности сервера можно использовать специальные представления с префиксом V$. Например, сведения о текущих сессиях можно получить, обратившись к представлению V$SESSION.

После создания базы данных и ее запуска, требуется, используя учетные записи SYS или SYSTEM, войти в СУБД для создания учетных записей других пользователей. Пароли пользователей SYS и SYSTEM, а также других пользователей, установленных по умолчанию, рекомендуется изменить. Работать с данными с учетными записями SYS и SYSTEM не рекомендуется.

Пользователь (USER) — объект, обладающий возможностью создавать и использовать другие объекты Oracle, а также запрашивать выполнение функций сервера. К числу таких функций относится организация сессии, изменение состояния базы данных и др. С пользователем Oracle связана схема (SCHEMA), которая является логическим набором объектов базы данных, таких, как таблицы, последовательности, хранимые программы, принадлежащие этому пользователю. Схема имеет только одного пользователя-владельца, ответственного за создание и удаление этих объектов. При создании пользователем первого объекта неявно создается соответствующая схема. При создании им других объектов они по умолчанию становятся частью этой схемы. Для просмотра объектов схемы текущего пользователя можно использовать представление словаря данных USER_OBJECTS.

Завершая работу с базой данных, администратор размонтирует ее, отсоединяя от экземпляра, а затем останавливает экземпляр. В процессе останова сервера базы данных Oracle завершает запущенные процессы и закрывает файлы операционной системы, в которых хранится информация базы данных.

Информация о результатах операции

Выполнение какой-либо операции клиентского приложения или оператора SQL на сервере может завершиться с различными итогами. Успешность завершения операции определяется значениями возвращаемых ею параметров, характеризующих итог выполнения операции, или сообщениями об ошибках. Сообщения об ошибках специфичны для различных продуктов Oracle. Во всех сообщениях об ошибках указывается префикс, указывающий на то, какое приложение выдает ошибку. Например, префикс IMP указывает на то, что ошибка произошла при импорте данных с использованием утилиты импорта.

Для сервера Oracle ошибки имеют префикс ORA, а соответствующие параметры передаются в форме двух кодов завершения: SQLCODE, SQLERRM. Значение SQLERRM содержит развернутое текстовое сообщение об ошибке с указанием ее причин. Пятисимвольное значение параметра SQLCODE определяется международным стандартом, формируемым организацией ISO. Нулевое значение параметра свидетельствует об успешном завершении операции, отрицательные значения соответствуют завершению операции с некоторой ошибкой, а положительные – завершению с предупреждением о возможной ошибке или предназначены для передачи параметров, характеризующих особенности завершения операции. Список типов ошибок сервера Oracle включает в себя несколько тысяч типов. Они обеспечивают точную диагностику при самых различных программных ошибках.

Пример:

SQL Error: ORA-00942: table or view does not exist

Поддержка мультиязычности в Oracle

Для корректной работы приложений с русским языком требуется специальная настройка. Поддержка национальных языков Oracle (National Language Support - NLS) позволяет взаимодействовать с сервером на различных языках, поддерживая различные схемы кодирования символов. Oracle поддерживает как однобайтовые, так и мультибайтовые схемы.

Имеется несколько настроек, связанных с языковыми особенностями. Они определяют используемый по умолчанию формат дат, чисел и других свойств обработки данных.

В базе данных имеется 3 уровня для настройки NLS: база данных, экземпляр и сессия. В настройки NLS входят NLS_LANGUAGE, NLS_TERRITORY, NLS_CHARSET, NLS_DATE_FORMAT и др.

Настройки NLS базы данных переписываются настройками NLS экземпляра, настройки NLS базы данных и экземпляра переписываются настройками NLS сессии.

Просмотреть текущие значения параметров NLS можно в системном представлении V$NLS_PARAMETERS.

Настройки NLS сессии. Просмотр SELECT * from NLS_SESSION_PARAMETERS. Изменение – с помощью оператора ALTER SESSION. Например: ALTER SESSION set NLS_DATE_FORMAT = 'DD/MM/YYYY'.

Настройки NLS экземпляра. Просмотр SELECT * from NLS_INSTANCE_PARAMETERS. Изменение – с помощью оператора ALTER SYSTEM.

Настройки NLS базы данных. Просмотр SELECT * from NLS_DATABASE_PARAMETERS. Параметры задаются на этапе создания б.д. и не изменяются в дальнейшем. Настройке также подлежат язык вывода сервером информационных сообщений, чисел, дат и т.п.

Следующий оператор выполняется корректно лишь в том случае, если используется язык Russian и задан подходящий формат даты по умолчанию:

SELECT * FROM TAB1

WHERE AT1 > ’20-ФЕВ-2001’.

Чтобы сделать оператор независимым от параметров NLS, следует использовать функции преобразования типов, например,

SELECT * FROM TAB1

WHERE AT1 > TO_DATE(’20-02-2001’, ‘DD-MM-YYYY’).

В завершение отметим, что если необходимо организовать хранение в одной базе данных информации на нескольких языках, то требуется каким-либо способом ее помечать, отведя для данных на различных языках различные таблицы или вводя в них дополнительный столбец, значение которого указывает на язык значений остальных столбцов.

SQL в СУБД Oracle

SQL является фактическим стандартом языковых средств обработки данных для современных СУБД. SQL является языком четвертого поколения. Это означает, что он описывает то, что нужно выполнить, а не то, как это должно быть сделано.
Например,

```
Delete From students
  Where major = ‘Nutrition’;
```

(таблица students создана следующим образом:

```
Create Table students(
   Id number(5) primary key,
   First_name Varchar(20),
   Last_name Varchar(20),
   Major Varchar(30),
   Current_credits number(3)
);
)
```

В данном примере не известно, как база данных найдет студентов, специализирующихся по питанию. Чтобы узнать, какие записи необходимо удалить, сервер, скорее всего, просмотрит сведения обо всех студентах, однако реальную картину этой операции пользователи не видят.

Языки третьего поколения, например C и COBOL, по природе своей процедурны. Программа, составленная на языке третьего поколения (3GL, third — generation language), реализует пошаговый алгоритм решения проблемы. Например, можно выполнить операцию DELETE примерно таким образом:

```
LOOP (циклически просмотреть) запись о каждом студенте
  IF (если) для этой записи major = 'Nutrition' THEN
    DELETE (удалить) эту запись;
  END IF (конец если);
END LOOP (конец цикла);
```

Объектно-ориентированные языки программирования, такие как C++ и Java, также являются языками третьего поколения. Хотя в них реализуются принципы объектно-ориентированного программирования, алгоритмы все еще остаются пошаговыми.

Операторы SQL можно разбить на несколько групп: операторы манипулироавия данными DML (Data Manipulation Language), операторы определения данных DDL (Data Definition Language), группа операторов управления транзакциями TCL (Transaction Control Language) и группа операторов предоставления доступа DCL (Data Control Language). Операторы DDL предназначены для создания, модификации и удаления объектов баз данных.

Oracle поддерживает стандарт ANSI (American National Standards Institute — Американский национальный институт стандартов) для языка SQL, как определено документом ANSI/ISO/IEC 9075 (1-4, 9-11, 13, 14): 2008 "Database Language SQL". Этот стандарт, известный как SQL99 (или SQL2), определяет только язык SQL и не описывает SGL-расширения, которые предлагаются в PL/SQL. Oracle 11g поддерживает большинство свойств, требуемых частью CORE стандарта. Дополнительная информация находится в руководстве Oracle SQL Reference.

Язык описания данных Oracle

К числу основных функций языка описания данных можно отнести:

— идентификацию типов данных;

— присвоение уникальных имен различным типам данных;

— спецификацию структуры объекта базы данных;

— спецификацию ключей.

В число дополнительных функций языка описания данных можно включить:

— определение частных характеристик элементов данных;

— определение ограничений целостности;

— описание элементов физического уровня хранения данных.

Под частными характеристиками элементов данных обычно понимается определение длины символьных строк, масштаба и точности числовых данных и т. п. Обратите внимание, что формат хранения не обязательно совпадает с форматом, в котором пользователь получает данные. Сервер Oracle поддерживает возможность переформатирования значений данных.

Указание ограничений целостности используется для повышения достоверности ввода данных. Автоматический контроль хорошо структурированных данных повышает достоверность информации в базе данных.

Типы данных Oracle

Любой отдельно взятый тип является либо определяемым системой (т.е. встроенным), либо определяемым пользователем.

Для описания пользовательского типа в SQL используется оператор CREATE TYPE. При описании пользовательского типа добавляется его определение в словарь данных.

Например:

```
CREATE TYPE customer_typ_demo AS OBJECT
    ( customer_id        NUMBER(6)
    , cust_first_name    VARCHAR2(20)
    , cust_last_name     VARCHAR2(20)
    , cust_address       CUST_ADDRESS_TYP
    ) ;
```

Для удаления используется оператор DROP TYPE. При удалении соответствующее определение из словаря данных удаляется.

DROP TYPE customer_typ_demo FORCE;

Ключевое слово FORCE позволяет удалить тип, даже если от него зависят объекты базы данных. Столбцы таблицы с удаленным типом данных помечается UNUSED и становятся недоступными.

Вообще говоря, при описании нового типа данных необходимо также определить различные операторы над этим типом.

Кроме того, любой отдельно взятый тип может быть либо скалярным, либо нескалярным. Определения этих понятий приведены ниже.

- Нескалярным (nonscalar) называется тип, значения которого явно определены как имеющие множество видимых пользователю, непосредственно доступных компонентов. В частности, в этом смысле являются нескалярными типы отношения, поскольку отношения имеют такие видимые пользователю компоненты, как кортежи и атрибуты.
- Скалярным (scalar) называется тип, который не является нескалярным.

Пользовательские типы могут быть как скалярными, так и нескалярными. Нескалярными могут структурированные типы, а также типы, созданные с помощью генераторов типов REF, ROW, ARRAY (!!!) (генератор типа – это оператор, который возвращает тип).

Все типы данных, перечисленных в стандарте ANSI SQL92, полностью поддерживаются в Oracle.

Поддерживается строгая типизация, но в ограниченном объеме. К встроенным типам можно применить определенный способ классификации, согласно которому они подразделяются на 10 отдельных категорий:

(проверить классификацию!!!)

- истинностное значение;
- битовая строка;
- двоичное значение;
- символьная строка;
- числовое значение;
- дата;
- время;
- временная отметка;
- интервал год / месяц;
- интервал сутки / время суток.

Строки символов

Тип CHARACTER используется для хранения строк фиксированной длины. Для хранения строк резервируется определенное в параметре длина пространство. При необходимости короткая строка дополняется пробелами.

Синтаксис: CHARACTER [(длина)], CHAR[(длина)].

Если длина строки не указана явно, она полагается равной 1.

Максимальное значение параметра длина — 255 символов.

Пример

Strl CHAR(15)

Str2 CHARACTER

Тип VARCHAR используется для хранения строк переменной длины. Для хранения строк резервируется реально необходимое пространство.

Синтаксис: VARCHAR [(длина)], CHAR VARYING [(длина)], CHARACTER VARYING [(длина)].

Если длина строки не указана явно, она полагается равной 1.

Пример

Varstrl VARCHAR(15)

Varstr2 CHAR VARYING(7)

Тип VARCHAR2 [Только для Oracle] используется для хранения строк переменной длины. Для хранения строк резервируется реально необходимое пространство. Основная причина введения типа VARCHAR2 состоит в том, что фирма-производитель декларирует неизменность этого типа в более поздних реализациях Oracle, в то время как тип VARCHAR будет соответствовать требованиям стандартов SQL.

Синтаксис: VARCHAR2 (длина). Длина строки должна быть указана явно.

Пример

.. Varstr3 VARCHAR2(15)

По умолчанию в SQL все строковые литералы имеют тип CHAR.

Для хранения символьных данных с использованием национальных алфавитов предназначены типы NCHAR и NVARCHAR2.

Числовые типы

Тип INTEGER используется для представления целых чисел в диапазоне от -2^{31} до 2^{31}.

Синтаксис: INTEGER, INT.

Пример

Varintl INTEGER

Varint2 INT

Тип NUMBER [Только для Oracle] используется для представления чисел с заданной точностью.

Синтаксис: NUMBER [(точность [, масштаб])].

Точность – количество цифр в числе. Масштаб – количество цифр после запятой (или округление). Если значение параметра точность не указано явно, оно полагается равным 38. Значение параметра масштаб по умолчанию предполагается равным 0. Значение параметра точность может изменяться от 1 до 38, значение параметра масштаб может изменяться от -84 до 127. Использование отрицательных значений масштаба означает сдвиг десятичной точки в сторону старших разрядов. Например, определение NUMBER (7, -6) означает округление до миллионов.

Пример

Varnumber NUMBER

Number (7, 2) значения от -99999,99 до 99999,99

Number (7, 9) значения от -0,009999999 до 0,009999999

Number (7, -2) значения от -999999900 до 999999900.

Для совместимости с другими СУБД Oracle поддерживает типы данных DECIMAL, DOUBLE_PRECISION, NUMERIC, DEC и REAL.

Типы DECIMAL И NUMERIC полностью эквивалентны типу NUMBER.

Синтаксис: DECIMAL [(точность [, масштаб])],

DEC [(точность [, масштаб])], NUMERIC [(точность [,масштаб])].

Пример

Vardecl DEC

Vardec2 DEC(15)

Vardec3 DECIMAL (8, 3)

Varnum NUMERIC

Тип ROWID

ROWID — специальный тип данных, который служит для представления указателей на запись в таблице. Каждая строка в базе данных имеет некоторый адрес. Этот адрес находится в псевдостолбце ROWID и имеет тип ROWID. При создании строки в таблице ей сразу присваивается ROWID, который остается неизменным до ее удаления или реорганизации данных. Использование ROWID — самый быстрый способ доступа к строке в таблице.

SQL > SELECT ROWID FROM Schedules;

Так как значение ROWID является уникальным для любой строки в таблице, ее можно использовать, например, для удаления повторяющихся строк в таблице. Пример такой операции приведен в следующем листинге.

```
SQL> CREATE TABLE Tabl (Atl NUMBER, At2 NUMBER);
Table created.

SQL> INSERT INTO Tabl VALUES(1, 2);
1 row created.
SQL> INSERT INTO Tab1 VALUES(1, 2);
1 row created.

SQL> DELETE FROM Tab1
2 WHERE ROWID NOT IN (SELECT MIN(ROWID)
3 FROM Tabl
4 GROUP BY Atl, At2;
5 )
1 row deleted.
```

Обычно, значение rowid однозначно идентифицирует строку в базе данных (однако, строки в различных таблицах и хранящихся в одном кластере могут иметь одинаковый rowid).

Rowid используется в следующих случаях:

1. Использование rowid - самый быстрый способ доступа к строке.

2. Rowid может показать, каким образом строки хранятся в таблице.

3. Они являются уникальными идентификаторами строк в таблице.

Однако не рекомендуется использовать rowid в качестве первичного ключа. Если вы удалите и вставите вновь строку с помощью утилит экспорта и импорта, rowid может измениться. Если вы удалите строку, Oracle может вновь использовать rowid этой строки для новых строк.

Rowid фактически не хранится в базе данных, это псевдостолбец. Значение можно использовать в SELECT. Однако вы не можете вставлять, изменять или удалять значение в этом псевдостолбце.

Битовые строки

Тип RAW [Только для Oracle] используется для хранения двоичных строк переменной длины. Отличие типа RAW от типов CHAR, VARCHAR2 состоит в том, что для типов символьных строк Oracle производит автоматическое преобразование данных при их передаче между клиентом и сервером.

Oracle выдает данные типа RAW в шестнадцатеричном виде.

Синтаксис: RAW [(длина)]. Параметр длина измеряется в байтах. Максимальное значение параметра длина — 2000 байт.

Пример

VarRaw RAW(10)

Тип LONG RAW [Только для Oracle] используется для хранения больших битовых строк переменной длины.

Синтаксис: LONG RAW [(длина)]. Параметр длина измеряется в байтах. Если длина строки не указана явно, она полагается равной 2 мегабайтам. Максимальное значение параметра длина — 2 гигабайта символов. Для переменных типа LONG RAW невозможно построение индекса.

Пример

VarLongRaw LONG RAW(1000000)

Дата и время

Тип DATE [Только для Oracle] используется для хранения даты и времени. Допускаются даты с 1 января 4712 г. до н.э. до 31 декабря 4712 г. н.э. Для формирования значения типа DATE в SQL и PL/SQL обычно используется встроенная функция TO_DATE('символьная_строка_даты', 'формат_даты’). При определении даты без уточнения времени по умолчанию принимается время полуночи.

Синтаксис: DATE.

Пример

VarDate DATE

Наличие специального типа для хранения даты и времени позволяет поддерживать специальную арифметику дат и времен. Добавление к переменной типа DATE целого числа интерпретируется Oracle как определение более поздней даты, а вычитание выполняется как определение более ранней.

Также возможно использование юлианской даты. Юлианская дата — это число дней, прошедших с 1 января 4712 г. до нашей эры. Реализация юлианской даты в Oracle не имеет компоненты времени. Для использования юлианской даты в функции TO_DATE применяется маска формата "J".

```
SQL> SELECT TO_CHAR(TO_DATE('01-01-2002',
2 'DD-MM-YYYY’), 'J') JDATE FROM dual;
JDATE
2452276
```

Время хранится с точностью до секунды. Когда нужно организовать обработку дат с более высокой точностью, то можно прибегнуть к различным хитростям, например, хранить наносекунды в числовом поле, написав библиотеку функций для обработки его значений и

используя эти функции в запросах. Или использовать специализированные пакетные функции, работающие с сотыми долями секунды.

LOB-объекты

Тип BLOB [Только для Oracle] используется для хранения двоичных данных размером до 4 гигабайт. Для работы с большими двоичными объектами используется стандартный пакет DBMS_LOB, о котором рассказывается позже.

Синтаксис: BLOB.

Пример

VarBlob BLOB

Тип CLOB [Только для Oracle] используется для хранения символьных данных переменной длины размером до 4 гигабайт, использующих однобайтовую кодировку.

Синтаксис: CLOB.

Пример

VarCClob CLOB

Тип NCLOB [Только для Oracle] используется для хранения символьных данных размером до 4 гигабайт, использующих одно- или многобайтовую кодировку.

Синтаксис: NCLOB.

Пример

VarNClob NCLOB

Тип BFILE [Только для Oracle] используется для хранения указателей на двоичные данные, находящиеся во внешних по отношению к СУБД файлах. Сами файлы хранятся в файловой системе.

Синтаксис: BFILE.

Пример

VarBFile BFILE

Создание и удаление таблиц

Существуют требования к именованияю таблиц и столбцов:

- начинается с буквы;
- длина: 1-30 символов;
- содержит только следующие символы: A-Z, a-z, 0-9, _, $, #;
- не повторяет название другого объекта, принадлежащего тому же пользователю;
- не является зарезервированным словом Oracle.

Названия не чувствительны к регистру. Чтобы название стало чувствительным к регистру, его необходимо заключить в “”.

Оператор определения таблиц Oracle содержит довольно большое число ключевых слов и параметров. Рассмотрим сокращенное множество конструкций.

```
CREATE TABLE [имя_схемы.] имя_таблицы
([ограничение_целостности_таблицы | имя_столбца
тип_данных_столбца [ DEFAULT выражение]
[ограничение_целостности_столбца ... ] ]
[,{ограничение_целостности_таблицы | имя_столбца
тип_данных_столбца [ DEFAULT выражение]
[ограничение_целостности_столбца ... ] }]... )
[{{PCTFREE целое | PCTUSED целое | INITRANS целое |
MAXTRANS целое |
TABLESPACE имя_табличной_области |
STORAGE размер_памяти }…}]
[PARALLEL возможность_параллельной_обработки ]
[{ENABLE проверяемые_ограничения_целостности |
DISABLE игнорируемые_ограничения_целостности}...]
```

[AS запрос]
[CACHE | NOCACHE]

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

Ключевые слова PCTFREE, PCTUSED, INITRANS, MAXTRANS, TABLESPACE, STORAGE характеризуют пространство, распределяемое при работе с таблицей.

Ключевое слово PCTFREE определяет процент пространства блока, который резервируется для нужд модификации данных таблицы. Допустимые значения от 0 до 99. Значение по умолчанию 10. То есть, если данный параметр не указан, то при заполнении каждого блока 10% пространства остается неиспользованным. Это пространство используется для записи в него данных при выполнении в дальнейшем операций модификации строк таблицы.

Параметр PCTUSED задает нижнюю границу, достижение которой вызывает возврат блока данных в список свободных областей. Допустимые значения от 1 до 99. Значение по умолчанию 40. То есть, если в блоке занято менее 40% пространства в него вводятся данные при выполнении операции вставки.

Как только блок данных будет заполнен до процента PCTFREE, в этот блок невозможно будет вставить новые строки до тех пор, пока процент памяти, используемой в этом блоке, не упадет ниже значения параметра PCTUSED.

Оба параметра настраиваются в паре. Устанавливая разные варианты значений для этих параметров, можно оптимизировать использование дискового пространства. При настройке PCTFREE и PCTUSED необходимо помнить о двух ограничения. Во-первых, их сумма не может превышать 100. Во-вторых, PCTFREE нельзя устанавливать равным 0, так как это вызовет проблемы распределения памяти для внутренних операций.

Примеры использования.

1. Большая часть запросов содержит операторы UPDATE, которые увеличивают размеры записей.

PCTFREE = 20
PCTUSED = 40

PCTFREE установлен в 20, чтобы оставить достаточно места для записей, увеличивающихся в размере при обновлении. PCTUSED оставлен по умолчанию.

2. В основном запросы состоят из операторов INSERT и DELETE, а операторы UPDATE в среднем не увеличивают размер записи.

PCTFREE = 5
PCTUSED = 60

PCTFREE установлен в 5, так как в основном длины записей не изменяются. PCTUSED установлен в 60, чтобы избежать дополнительного выделения большого числа блоков данных, так как память, освобождаемая оператором DELETE, почти сразу же используется оператором INSERT.

3. Данные из таблицы выбираются в основном на чтение.

PCTFREE = 5
PCTUSED = 90

PCTFREE установлен в 5, так как операторы UPDATE используются редко. PCTUSED установлен в 90, так что для хранения данных используется большая часть блока. В результате уменьшается общее число используемых блоков.

Ключевое слово INITRANS определяет начальное число параллельных транзакций, которые могут выполняться для модификации данных блока. Значение по умолчанию 1. Ключевое слово MAXTRANS определяет максимальное число параллельных транзакций,

которые могут выполняться для модификации данных блока. В большинстве случаев явное задание этих параметров не требуется.

Ключевое слово TABLESPACE определяет имя табличной области, в которой будет размещена таблица. Если значение параметра не определено, то таблица размещается в табличной области, заданной по умолчанию для пользователя, который является владельцем схемы, содержащей таблицу.

Ключевое слово STORAGE определяет объем внешней памяти, выделяемый под таблицу. Для больших таблиц целесообразно явно выделять требуемую память для уменьшения запросов на динамическое выделение пространства для таблицы.

PARALLEL – задает уровень параллелизма для таблицы по умолчанию для операторов DML. Варианты: NOPARALLEL – параллелизм выключен. PARALLEL – уровень параллелизма определяется сервером автоматически. PARALLEL integer – устанавливает уровень параллелизма равным integer (количество потоков, используемых при параллельной обработке). В большинстве случаев integer указывать нет необходимости.

Ключевое слово ENABLE указывает на включение ограничений целостности для данной таблицы. Соответствующее ограничение целостности должно быть определено в данном предложении создания таблицы. По умолчанию все ограничения целостности, определенные в предложении, включаются.

Ключевое слово DISABLE указывает на выключение ограничений целостности для данной таблицы. Соответствующее ограничение целостности должно быть определено в данном предложении создания таблицы.

Конструкция AS запрос включает в создаваемую таблицу строки, являющиеся результатом выполнения запроса. Обратите внимание на необходимость определенной осторожности при использовании вставки строк через подзапрос и определение ограничений целостности в том же предложении. (Если результат запроса не соответствует ограничениям целостности, то Oracle не создает таблицу и возвращает сообщение об ошибке.)

Ключевое слово CACHE указывает на то, что блоки, выбираемые из таблицы, помечаются в системном кэше, как наиболее используемые. Рекомендуется для маленьких таблиц, используемых для преобразований кодов в значения. По умолчанию используется значение NOCACHE, для которого выбранные блоки помещаются в конец таблицы частот обращений к кэшу.

Существующие таблицы могут быть модифицированы с помощью команды ALTER TABLE. С ее помощью можно добавить один или несколько новых столбцов, ограничения целостности, модифицировать определение существующего столбца (тип данных, длину, умалчиваемое значение или ограничение целостности NOT NULL), модифицировать параметры хранения и транзакций (PCTFREE, PCTUSED, INITRANS, MAXTRANS и т.д.).

Добавление столбца:

```
(ALTER TABLE my_employees
  ADD (Age NUMBER(3)));
```

Добавление ограничения целостности:

```
ALTER TABLE my_employees
  ADD CONSTRAINT ...
```

Удаление ограничения на email:

```
ALTER TABLE my_employees
  DROP UNIQUE (email);
```

Переименование таблицы

```
(ALTER TABLE my_employees
  RENAME to my_employees_tmp;)
```

Для удаления из базы данных таблицы (вместе с ее содержимым) используется оператор DROP TABLE. Для выполнения операции уничтожения таблицы необходимо быть либо

владельцем таблицы, либо иметь привилегию DROP ANY TABLE. Когда таблица уничтожается, все блоки становятся свободными для использования под данные или индексы других таблиц. Оператор удаления таблицы Oracle использует следующий синтаксис:

DROP TABLE [имя_схемы.]имя_таблцы [CASCADE CONSTRAINTS].

Все индексы и триггеры, ассоциированные с таблицей, даже если они были созданы другим пользователем, удаляются. Все хранимые программы, зависящие от таблицы, остаются, но становятся недействительными (непригодными для использования). Все синонимы удаленной таблицы остаются, но возвращают ошибку при обращении к ним. Представления, синонимы и программы вновь становятся актуальными, если таблица создается заново (после их перекомпиляции).

Если указано ключевое слово CASCADE CONSTRAINTS, то удаляются все ограничения целостности, ссылающиеся на первичные и уникальные ключи данной таблицы. Если такие ссылки существуют, CASCADE CONSTRAINTS отсутствует, то удаление таблицы не выполняется и сервер возвращает сообщение об ошибке. Перед удалением таблицы рекомендуется определить через представление словаря данных USER_CROSS_REFS зависимости других таблиц от данной таблицы.

Средства определения и уничтожения представлений

Представление — это поименованная, динамически поддерживаемая сервером выборка из одной или нескольких таблиц или других представлений. Оператор SELECT, определяющий выборку, ограничивает видимые пользователем данные. Кроме того, представление позволяет эффективно ограничить данные, которые пользователь может модифицировать. Используя представления, администратор базы данных ограничивает доступную пользователям часть логического пространства базы данных только теми данными, которые реально им необходимы. Оператор определения представлений Oracle использует следующий синтаксис:

CREATE [OR REPLACE] [{FORCE | NO FORCE}] VIEW
[имя_схемы.] имя_представления
AS запрос WITH { READ ONLY | CHECK OPTION
[CONSTRAINT ограничение_целостности]}

Ключевые слова OR REPLACE указывают на принудительное замещение старого представления новым. Использование этого параметра позволяет не выполнять повторного предоставления привилегий, которое было бы необходимо, если использовать команды DROP VIEW и CREATE VIEW для уничтожения и создания представления заново.

Ключевое слово FORCE указывает на принудительное создание представления вне зависимости от того, существуют ли базовые таблицы представления, и есть ли у пользователя, создающего представление, привилегии на выборку из базовых таблиц.

Конструкция NO FORCE (используемая по умолчанию) указывает на обязательность существования базовых таблиц и представлений и наличия у пользователя, создающего представление, привилегий на доступ к базовым таблицам. При нарушении какого-либо из этих условий представление не создается.

Параметр запрос используется для обозначения любого синтаксически правильного запроса, не содержащего ключевого слова ORDER BY или конструкции FOR UPDATE.

Ключевое слово WITH READ ONLY указывает на запрещение для базовых таблиц операций модификации данных с указанием представления.

Ключевое слово WITH CHECK OPTION указывает на то, что строки в базовых таблицах, изменяемые и вставляемые через представление, должны соответствовать критерию отбора в запросе, определяющем представление.

Требование WITH CHECK OPTION в определении представления имеет смысл только в случае определения изменяемой представляемой таблицы, которая определяется спецификацией запроса, содержащей раздел WHERE. При наличии этого требования не допускаются изменения представляемой таблицы, приводящие к появлению в базовых таблицах строк, не видимых в представляемой таблице (то есть таких строк, которые не

удовлетворяют условию поиска раздела WHERE спецификации запроса). Если WITH CHECK OPTION в определении представления отсутствует, такой контроль не производится.

Ключевое слово CONSTRAINT определяет имя ограничения, используемое для проверки.

Представление является изменяемым, то есть по отношению к нему можно использовать оператор DELETE, INSERT и UPDATE, в том случае, если выполняются следующие условия для образующего представление запроса:

— в списке выборки не указано ключевое слово DISTINCT;

— каждое арифметическое выражение в списке выборки представляет собой одну спецификацию столбца, и спецификация одного столбца не появляется более одного раза;

— в условии выборки раздела WHERE не используются подзапросы;

— в запросе отсутствуют конструкции GROUP BY и HAVING.

Если в списке выборки спецификации запроса имеется хотя бы одно арифметическое выражение, состоящее не из одной спецификации столбца, или если имя хотя бы одного столбца участвует в списке выборки более одного раза, определение должно содержать список имен столбцов таблицы. Более просто, нужно явно именовать столбцы представляемой таблицы, если эти имена не наследуются от столбцов таблиц раздела FROM спецификации запроса.

Для того, чтобы конструктивно работать с представлением, пользователь должен, как минимум, иметь привилегию SELECT для всех таблиц, которые участвуют в запросе, формирующем представление. Поэтому привилегии, которыми обладает пользователь на базовые таблицы, наследуются представлением для пользователя, который его создает. Если пользователь обладает любой комбинацией привилегий INSERT, UPDATE, DELETE для базовых таблиц, то эти привилегии будут автоматически наследоваться представлением. В то же время пользователь, не имеющий привилегий на модификацию строк базовых таблиц, не может получить соответствующие привилегии в представлении.

Еще одно полезное свойство представлений состоит в том, что они позволяют реализовать доступ пользователей к данным, которые являются производными от данных базовых таблиц.

Удаление представления выполняется командой DROP VIEW. Для удаления представления необходимо быть его владельцем или иметь привилегию DROP ANY VIEW. Используется следующий синтаксис:

DROP VIEW [имя_схемы.] имя_представления

При удалении представления объекты, ссылающиеся на удаляемое представление, не уничтожаются, а становятся недействительными. Привилегии на удаляемое представление также отменяются.

Ограничения целостности

Ограничения целостности представляют собой автоматическую поддержку некоторой системы правил, описывающих допустимость и достоверность хранимых и вводимых значений. Кроме того, ограничения целостности могут предотвратить удаление таблицы в случае наличия зависимостей.

Ограничения целостности определяются как часть определения таблицы и хранятся в словаре данных. Таким образом, все приложения, получающие доступ к таблице, должны удовлетворять этим ограничениям. Когда вы изменяете эти ограничения, изменения необходимо сделать один раз – на уровне базы данных, а не на уровне приложений. Ограничения целостности позволяют гарантировать, что требования к данным будут соблюдаться независимо от способа их загрузки или изменения.

Ограничения целостности на уровне столбца:

column [CONSTRAINT constraint_name] constraint_type, …

Ограничения целостности на уровне таблицы:

column, …

[CONSTRAINT constraint_name] constraint_type (column, …), …

Ограничение целостности на уровне таблицы может охватывать несколько столбцов.

Рассмотрим предусмотренные в Oracle типы статических ограничений целостности:

— ограничение на определенность значения атрибута (NOT NULL);

— ограничение на уникальность значения атрибутов (UNIQUE);

— ограничение — первичный ключ (PRIMARY KEY);

— ограничение — внешний ключ (FOREIGN KEY);

— ограничение целостности, задаваемое предикатом (CHECK).

Ограничения связываются с конкретной таблицей либо в момент ее создания, либо создаются оператором ALTER TABLE. Ограничения целостности могут быть поименованы с помощью ключевого слова CONSTRAINT. Именование ограничений обычно используется для локализации нарушенного ограничения в большой системе. Система всегда присваивает некоторое имя каждому ограничению целостности, но рекомендуется явно назначать унифицированные имена ограничениям, скажем, первичные ключи именовать PK_имя_таблицы.

NOT NULL

Наличие ограничения NOT NULL для некоторого атрибута приводит к автоматическому запрещению изменений или вставок строк, содержащих неопределенные значения этого атрибута. Это ограничение может быть создано при создании таблицы.

Задается только на уровне столбцов.

UNIQUE

Если множество атрибутов (или атрибут) описаны, как уникальные, сервер запретит повторные значения в столбце или наборе столбцов.

ALTER TABLE <TableName>
ADD CONSTRAINT <UniqueName> UNIQUE
(<ColumnName>, …)
ENABLE;

Може определяться как на уровне столбцов (для уникальности отдельных столбцов), так и на уровне таблицы (может включать уникальность совокупности столбцов).

PRIMARY KEY

Ограничение — первичный ключ описывает совокупность атрибутов или атрибут, который для данной таблицы выбран в качестве первичного ключа. Отметим, что в таблице может быть только один первичный ключ. Первичный ключ может состоять как из одного столбца, так и из нескольких, т.е. быть составным. При выполнении вставки новой записи или модификации атрибутов, входящих в состав первичного ключа, автоматически проверяется уникальность набора их значений и отсутствие неопределенных (NULL) значений атрибутов, входящих в состав первичного ключа.

Ограничение может определяться при создании таблицы, так и после, с помощью оператора ALTER TABLE.

ALTER TABLE <TableName>
ADD CONSTRAINT <KeyName> PRIMARY KEY
(<ColumnName>, …) ENABLE;

При создании:

```
CREATE TABLE employee(
id NUMBER(6) CONSTRAINT pk_employee_id PRIMARY KEY,
first_name VARCHAR2(20));

CREATE TABLE employee(
id NUMBER(6),
first_name VARCHAR2(20),
CONSTRAINT pk_employee_id PRIMARY KEY(id));
```

FOREIGN KEY

Ограничение — внешний ключ связывает значения наборов атрибутов двух таблиц: базовой (PARENT TABLE) и производной (CHILD TABLE). В частном случае одна таблица может быть и базовой и производной одновременно. Ограничения, определяемые внешними ключами, обычно называют поддержкой ссылочной целостности.

Это ограничение может задаваться как на уровне столбца, так и на уровне таблицы.

Ссылочная целостность определяет соотношения между различными столбцами, когда значения в одном наборе столбцов должны соответствовать значениям другого набора столбцов. Если некоторый набор атрибутов производной таблицы объявлен внешним ключом (FOREIGN KEY), то для каждого его значения должна найтись запись базовой таблицы с тем же значением ключа.

ALTER TABLE <ChildTableName>
ADD CONSTRAINT <ForeignKeyName>
FOREIGN KEY (<ChildTableColumnName>)
REFERENCES <ParentTableName>
(<ParentTableColumnName>) ENABLE;

Ограничение ссылочной целостности не только определяет допустимые значения во внешнем ключе производной таблицы, но и действие при операции с первичным ключом базовой таблицы.

Например, ссылочное действие каскадного удаления ограничения — внешнего ключа определяет, что при удалении строки базовой таблицы должны удаляться все зависящие от нее строки производной таблицы.

Если по внешнему ключу не определены никакие дополнительные ограничения, то любое количество строк в подчиненной таблице может ссылаться на одно и то же значение первичного ключа. В этом случае устанавливается отношение "один ко многим". Когда по внешнему ключу определено ограничение UNIQUE, лишь одна строка в подчиненной таблице может ссылаться на данное значение первичного ключа. Таким образом устанавливается связь "один к одному" между первичным и внешним ключами. Для удаления таблицы, на строки которой ссылаются строки другой таблицы, требуется указание конструкции CASCADE CONSTRAINTS.

CHECK

Ограничение целостности, задаваемое предикатом, позволяет при вводе или модификации данных проверить принадлежность значения атрибута таблицы множеству допустимых значений или выполнение иного условия, задаваемого предикатом (логическим выражением). Синтаксис ограничения целостности, задаваемого предикатом, выглядит следующим образом:

CHECK (предикат)

Параметр предикат может быть любым предикатом, определенным на множестве атрибутов таблицы. Ограничение CHECK имеет следующие ограничения: оно должно быть выражением, вычисляемым над значениями вставляемой или обновляемой строки, не должно содержать подзапросов и ссылаться на последовательности.

ALTER TABLE <TableName>
ADD CONSTRAINT <ConstraintName>
CHECK (<Constraint>) ENABLE;

По умолчанию проверка ограничений целостности происходит непосредственно при выполнении действий над данными. Однако в Oracle допускается так называемая отложенная проверка ограничений, которая предусматривает проверку при фиксации транзакции.

Такая проверка удобна, например, при изменении значения первичного ключа для записи, у которой существуют подчиненные записи в другой таблице. В этом случае можно выполнить каскадное изменение значений ключей и зафиксировать транзакцию.

Для включения отложенной проверки ограничений требуется при определении таблицы указать следующую конструкцию:

CONSTRAINT . . . [{ [[NOT] DEFERRABLE]
[INITIALLY {IMMEDIATE|DEFERRED}]
|[INITIALLY {IMMEDIATE|DEFERRED}]
[[NOT] DEFERRABLE]}]

По умолчанию ограничения целостности создаются не откладываемыми. Для создания откладываемого ограничения при его объявлении надо использовать ключевое слово DEFERRABLE. Если транзакция не начинается с команды SET CONSTRAINTS, то для откладываемого ограничения Oracle использует стандартный алгоритм проверки. При этом данные проверяются на соответствие ограничению в конце каждой команды DML. Если при определении ограничения были использованы ключевые слова INITIALLY DEFERRED, данные проверяются на соответствие ограничениям при фиксации транзакции. Если при создании ограничения сочетаются NOT DEFERRABLE INITIALLY DEFERRED, то возникнет ошибка SQL Error: ORA-02447: cannot defer a constraint that is not deferrable.

При проведении массовых операций над данными иногда целесообразно некоторые ограничения целостности отключить, провести операции, а затем опять включить ограничения.

Рассмотрим пример создания таблиц Item и Item_Price.

У таблицы Item два атрибута ID и Name. Ограничение pk_Item_ID, указывает, что атрибут ID является первичным ключом, ограничение nn_Item_Name, указывает, что атрибут Name не допускает ввода неопределенных значений.

У таблицы Item_Price три атрибута: Item_ID, Price и PriceDate. Значение атрибута Item_ID является внешним ключом по отношению к первичному ключу таблицы Item.

Значение атрибута PriceDate по умолчанию есть текущая дата. Ограничение nn_Item_Price_Price, указывает, что атрибут Price не допускает ввода неопределенных значений. Ограничение UNIQUE означает, что каждая строка должна быть уникальной.

```
CREATE TABLE Item
(ID Integer CONSTRAINT pk_Item_ID PRIMARY KEY,
Name Varchar2(50) CONSTRAINT nn_Item_Name NOT NULL);

CREATE TABLE Item_Price
(Item_ID Integer CONSTRAINT fk_Item_Price_Item_ID REFERENCES Item(ID),
Price Number CONSTRAINT nn_Item_Price_Price NOT NULL,
PriceDate Date DEFAULT SYSDATE UNIQUE);
```

Если бы таблицу Item было необходимо создать в схеме пользователя usr, размещенной в табличной области table_data (которая должна быть создана заранее) с ассоциированным с первичным ключом индексом, размещенным в табличной области index_data, под таблицу резервировался начальный экстент в 100 килобайт и определялся экстент приращения в 50 килобайт, то SQL-запрос создания таблицы Item выглядел бы следующим образом:

```
CREATE TABLE usr.Item
(ID Integer CONSTRAINT pk_Item_ID PRIMARY KEY
USING INDEX TABLESPACE index_data,
Name Varchar2(50) CONSTRAINT nn_Item_Name NOT NULL)
TABLESPACE table_data
STORAGE (INITIAL 100K NEXT 50K);
```

Пример ограничения на определенность значения атрибута Name таблицы Item.

```
INSERT INTO Item VALUES (1 , 'Pen');
(1 row created)
INSERT INTO Item VALUES (2, Null);
(ERROR at line 1:
ORA-01400: cannot insert NULL into ("sys"."Item"."Name")
```

Далее представлен пример ограничения на уникальность значений атрибутов таблицы Item_Price.

```
INSERT INTO Item_Price VALUES(1, 23, '01.01.2009');
1 row created.
INSERT INTO Item_Price VALUES(1, 23, '01.01.2009');
ERROR at line 1:
ORA-00001: unique constraint (Sys.SYS_C004752) violated
```

Рассмотрим пример автоматической проверки ограничений, связанных с выбором атрибута в качестве первичного ключа в таблице Item. Попытка создать еще один первичный ключ отвергается системой. Попытка изменить значение атрибута ID так, чтобы значения первичных ключей совпадали, также отвергается системой.

```
ALTER TABLE Item
ADD PRIMARY KEY (ID);
ERROR at line 2:
ORA-02260: table can have only one primary key

INSERT INTO Item VALUES (2, ‘Pencil’);
1 row created.
UPDATE Item SET ID=1 WHERE ID=2;
ERROR at line 1:
ORA-00001: unique constraint (Sys.PK_Item_ID) violated
```

Ввод данных в таблицу Item_Price со значением внешнего ключа, совпадающим с одним из значений первичного ключа таблицы Item, выполняется успешно. Попытка ввести строку в таблицу Item_Price со значением атрибута Item_ID, которому не соответствует ни одно значение первичного ключа таблицы Item, приводит к сообщению об ошибке.

```
INSERT INTO Item_Price VALUES(15, 23, ’01.01.2009’);
ERROR at line 1:
ORA-02291: integrity constraint (Sys.FK_Item_Price_Item_ID) violated - parent key not found
```

Рассмотрим пример, иллюстрирующий каскадное удаление. Определим соответствующую модификацию таблицы Item_Price. Тогда удаление строки в базовой таблице приводит к автоматическому удалению строк в производной таблице.

```
ALTER TABLE Item_Price
DROP CONSTRAINT fk_Item_Price_Item_ID;
Table altered.

ALTER TABLE Item_Price
ADD CONSTRAINT "fk_Item_Price_Item_ID"
FOREIGN KEY(Item_ID) REFERENCES Item("ID") ON DELETE CASCADE;
Table altered.

INSERT_INTO Item_Price VALUES(2, 33, ’01.02.2009’);
1 row created.
INSERT INTO Item_Price VALUES(2, 34, ’01.03.2009’);
1 row created.
SELECT * FROM Item_Price;
(возвращается 2 строки)
DELETE FROM Item WHERE ID=2;
```

1 row deleted.
SELECT * FROM Item_Price;
no data found

Пример удаления таблицы, для которой существует зависимая таблица.
DROP TABLE Item;
ERROR at line 1:
ORA-02449: unique/primary keys in table referenced by foreign keys
DROP TABLE Item CASCADE CONSTRAINTS;
Table dropped.

Пример добавления ограничения целостности в таблицу Item. Попытка ввести данные, не удовлетворяющие ограничению целостности, отвергается системой. Ввод данных, для значений которых предикат принимает истинное значение, выполняется.

ALTER TABLE Item
ADD CONSTRAINT Item_Check CHECK (Name<>'XXX');
Table altered.
INSERT INTO Item VALUES (1, 'X');
1 row(s) inserted
INSERT INTO Item VALUES (1, 'XXX');
ERROR.at line 1:
ORA-02290: check constraint (Sys.Item_CHECK) violated

Ниже приведен пример выполнения отложенной проверки ограничений целостности приведен (необходимо выполнять в SQL*Plus).

Drop Table Item;
Table dropped.

CREATE TABLE Item
(ID Integer CONSTRAINT pk_Item_ID PRIMARY KEY INITIALLY DEFERRED DEFERRABLE,
Name Varchar2(50) CONSTRAINT nn_Item_Name NOT NULL);
Table created.

INSERT INTO Item VALUES(1,'Pen');
1 row created.
INSERT INTO Item VALUES(1, 'Pencil');
1 row created.
COMMIT;
ERROR at line 1:
ORA-02091:' transaction rolled back ORA-00001: unique constraint (Sys.SYS_C005885) violated

Последовательности.

Последовательностью называется объект базы данных, генерирующий неповторяющиеся целые числа. Полученные из последовательности числа обычно используются в качестве значений для первичных ключей.

Числа, создаваемые последовательностью, могут либо возрастать постоянно, либо только до определенного предела, либо, по достижении предела, начинать возрастание заново, с начального значения. Последовательность может создавать цепочки как увеличивающихся чисел, так и уменьшающихся.

Можно задавать также и приращение значений.

На одно предложение SQL генерируется лишь один новый номер; иными словами, если в данном предложении псевдостолбец NEXTVAL применительно к одной и той же последовательности встречается несколько раз, то лишь для первого обращения будет возвращен новый номер последовательности, а все остальные обращения в этом же предложении возвратят тот же самый номер.

Для создания последовательности требуется привилегия CREATE SEQUENCE или CREATE ANY SEQUENCE.

Оператор определения последовательности Oracle использует следующий синтаксис:

CREATE SEQUENCE
[имя_схемы.] имя_последовательности
[INCREMENT BY приращение]
[START WITH начальное_значение]
[MAXVALUE наиболынее_значение | NOMAXVALUE]
[MINVALUE наименьшее_значение | NOMINVALUE]
[CYCLE | NOCYCLE]
[CACHE число_элементов | NOCACHE]
[ORDER | NOORDER]

Параметр имя_ последовательности задает имя последовательности. Параметр имя_схемы указывает на схему, в которой определяется последовательность. Если владелец последовательности не указан явно, то подразумевается пользователь, выдавший команду CREATE SEQUENCE.

Ключевое слово INCREMENT BY определяет интервал между последовательными номерами. Если параметр приращение имеет отрицательное значение, то последовательность убывающая, если положительное — последовательность возрастающая. Допустимо любое целое число, не равное нулю. Значение по умолчанию 1 (возрастающая последовательность значений).

Ключевое слово START WITH через параметр начальное_значение задает первый генерируемый последовательный номер. Если ключевое слово не указано, то по умолчанию для возрастающих последовательностей начальный генерируемый последовательный номер равен значению параметра MINVALUE, а для убывающих последовательностей — MAXVALUE.

Ключевое слово MAXVALUE через параметр наиболъшее_значение задает максимальное значение последовательного номера, которое будет генерироваться. Параметр наибольшее_значение определяет верхнюю границу последовательности, которая может быть любым целым числом, с количеством знаков, не превышающим 28 цифр и большим, чем параметры начальное_значение и наименьшее_значение (если они заданы). Отсутствие верхней границы указывается ключевым словом NOMAXVALUE, которое определяет для убывающих последовательностей значение -1, а для возрастающих последовательностей —10^{27}.

Ключевое слово MINVALUE через параметр наименьшее_значение задает минимальное значение последовательного номера, которое будет генерироваться. Параметр наименьшее_значение определяет нижнюю границу последовательности, которая может быть любым целым числом, с количеством знаков, не превышающим 28 цифр, меньшим, чем параметры наибольшее_значение и начальное_значение (если значение параметров задано). Отсутствие нижней границы указывается ключевым словом NOMINVALUE, которое определяет для убывающих последовательностей значение -10^{26}, а для возрастающих последовательностей значение -1.

Ключевое слово NOCYCLE является значением, используемым по умолчанию, и предполагает завершение генерирования последовательных номеров по достижении конца последовательности. Любая попытка получить очередной элемент последовательности после этого приведет к ошибке.

Если при определении последовательности указан параметр CYCLE, то после достижения очередным членом последовательности значения параметра наибольшее_значение (для

возрастающих последовательностей) выдается значение параметра наименьшее_значение. Если параметры наибольшее_значение и наименьшее_значение не указаны, то используются их значения по умолчанию. Для убывающих последовательностей параметры меняются местами по смыслу понятия убывающей последовательности.

Ключевое слово CACHE указывает на использование техники предварительной подготовки элементов последовательности, что обеспечивает их быстрое получение при запросе. Число последовательных номеров, хранящихся в области кэша оперативной памяти, определяется параметром число_элементов ключевого слова CACHE. Кэширование последовательности обеспечивают более быструю генерацию элементов последовательности. Заполнение кэша для каждой данной последовательности происходит после запроса первого элемента этой последовательности. В случае краха системы все кэшируемые последовательные номера, не использованные в зафиксированных транзакциях, теряются. По умолчанию предполагается, что в памяти будут кэшироваться 20 последовательных элементов для каждой последовательности. Значение параметра число_элементов ключевого слова CACHE не должно превышать разницы между параметрами, задаваемыми ключевыми словами MAXVALUE и MINVALUE.

Ключевое слово ORDER обеспечивает генерацию последовательных элементов точно в порядке поступления запросов. В большинстве случаев независимо от того, указано ли ключевое слово ORDER, элементы последовательности генерируются в порядке поступления запросов. Одна последовательность может использоваться для генерации первичных ключей для нескольких таблиц. Если два пользователя одновременно обращаются к одной последовательности, номера для каждого пользователя могут иметь промежутки, так как из непрерывной последовательности попеременно получают номера оба пользователя. Каждый из двух пользователей не будет видеть последовательные номера, сгенерированные для другого пользователя. При генерации элемента последовательности счетчик элементов изменяется независимо от того, успешно или неуспешно завершена транзакция.

Для удаления последовательностей используется команда DROP SEQUENCE. Для выполнения данной операции необходимо быть владельцем последовательности либо иметь привилегию DROP ANY SEQUENCE.

Оператор удаления последовательностей Oracle использует следующий синтаксис:

DROP SEQUENCE [имя_схемы.]имя_последовательности

Одной из ситуаций, когда необходимо уничтожение последовательности, является повторный старт последовательности. При этом необходимо учитывать, что объекты, которые зависят от этой последовательности, станут непригодными для использования. Поэтому для приведения последовательности к требуемому состоянию (сдвига на требуемое число элементов) рекомендуется использовать более изощренные способы.

Псевдостолбец NEXTVAL используется для генерирования очередного номера из указанной последовательности.

Ссылка на NEXTVAL приводит к генерированию очередного номера. Обращение имеет следующий синтаксис:

имя_последовательности.NEXTVAL.

Псевдостолбец CURRVAL используется для ссылки на текущее значение последовательного номера. В текущем сеансе NEXTVAL должен быть использован хотя бы один раз до ссылки на CURRVAL. Обращение к CURRVAL имеет следующий синтаксис:

имя_последовательности.CURRVAL.

PL/SQL

У каждого типа языка программирования есть свои достоинства и недостатки. Языки четвертого поколения, подобные SQL, как правило, проще (по сравнению с языками третьего поколения) и содержат меньшее число команд. Кроме того, они изолируют пользователя от базовых структур данных и алгоритмов, реализуемых исполняющей системой. Однако в

некоторых случаях процедурные конструкции языков 3GL полезны для более точного описания программы. Именно для этого применяется PL/SQL, который объединяет мощь и гибкость SQL (языка 4GL) и процедурные конструкции языка 3GL.

PL/SQL означает Procedural Language/SQL (процедурный язык/SQL). Как видно из названия, PL/SQL расширяет возможности SQL, добавляя в него такие конструкции процедурных языков, как:

• Переменные и типы данных (как предварительно определенные, так и определяемые пользователями)
• Управляющие структуры, такие как условные операторы и циклы
• Процедуры и функции
• Объектные типы и методы.

Процедурные конструкции объединяются с Oracle SQL, что дает в результате структурированный и эффективный язык программирования.

Предположим, что требуется изменить профилирующую дисциплину некоего студента. Если записи о нем нет, нужно создать новую запись. Это можно сделать при помощи следующего кода PL/SQL:

```
DECLARE
/* Обьявим переменные, которые будут использоваться в SQL-операторах */
v_NewMajor VARCHAR2(10) := 'History';
v_FirstName VARCHAR2(10) := 'Scott';
v_LastName VARCHAR2(10) := 'Urman';
BEGIN
/* Обновим таблицу students */
UPDATE students
SET major = v_NewMajor
WHERE first_name = v_FirstName
AND last_name = v_LastName;
/* Проверим, найдена ли запись. Если нет, нужно ее ввести. */
IF SQL%NOTFOUND THEN
INSERT INTO students (ID, first_name, last_name, major)
VALUES (student_sequence.NEXTVAL, v_FirstName, v_LastNatne,
v_NewMajor);
END IF;
END;
```

(последовательность student_sequence создана следующим образом:

```
CREATE SEQUENCE student_sequence
  START WITH 10000
  INCREMENT BY 1;)
```

Блочная структура PL/SQL

Язык программирования PL/SQL разработан на основе языка третьего поколения Ada. Многие конструкции, применяемые в Ada, можно найти в PL/SQL. Одним из общих свойств этих языков является их блочная структура. Из Ada заимствованы также обработка исключительных ситуаций, синтаксис объявления процедур и функций, модули.

Базовой единицей PL/SQL является блок (block). Все программы PL/SQL строятся из блоков, которые могут идти один за другим (последовательно) либо вкладываться один в другой. Как правило, в программе каждый блок выполняет определенную логическую единицу работы, что помогает отделить друг от друга различные задачи.

В разделе объявлений размещаются объявления всех переменных, курсоров и типов, используемых данным блоком. В этом разделе могут быть объявлены также локальные процедуры и функции. Такие подпрограммы будут доступны только в пределах блока. В выполняемом разделе осуществляется работа блока. В этом разделе могут находиться как процедурные, так и SQL-операторы. Ошибки обрабатываются в разделе исключительных ситуаций. Содержащийся в нем программный код не будет выполняться, если не возникнет ни одной ошибки.

Разделы блока ограничиваются ключевыми словами DECLARE (объявить), BEGIN (начало), EXCEPTION (исключительная ситуация) и END (конец). Кроме того, в конце блока необходимо ставить точку с запятой, это синтаксическое правило обязательно для блока. Таким образом, структура анонимного блока имеет вид:

```
DECLARE
/* Раздел объявлений */
BEGIN
/* Выполняемый раздел */
EXCEPTION
/* Раздел исключительных ситуаций */
END;
```

Структура блока, состоящего только из выполняемого раздела, будет выглядеть так:

```
BEGIN
/* Выполняемый раздел */
END;
```

Блок с разделом объявлений и выполняемым разделом, но без раздела исключительных ситуаций будет выглядеть следующим образом:

```
DECLARE
/* Раздел объявлений */
BEGIN
/* Выполняемый раздел */
END;
```

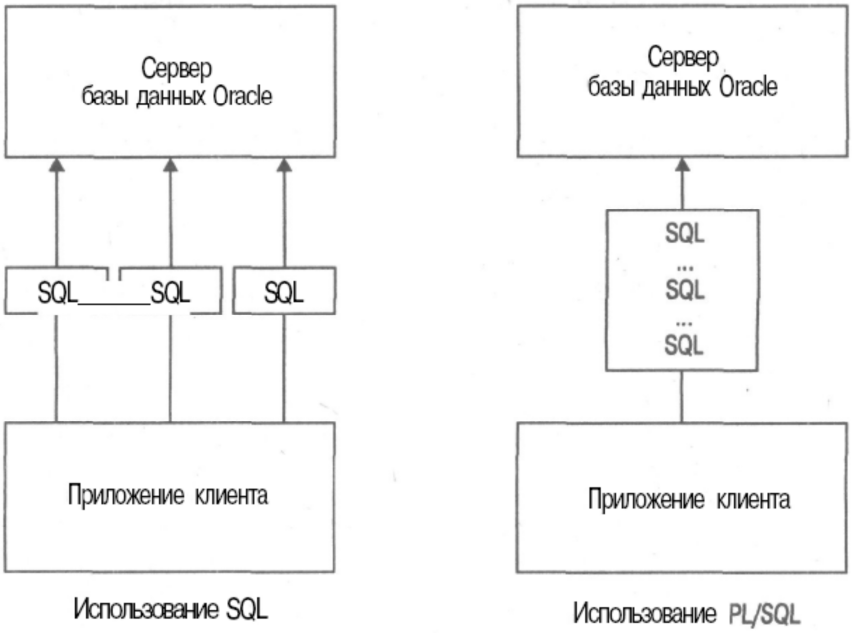
Существуют два различных типа блоков: анонимные блоки и именованные. Анонимные блоки обычно создаются динамически и выполняются только один раз. Этот тип блоков, как правило, создается в клиентской программе для вызова подпрограммы, хранящейся в базе данных. Именованные блоки отличаются тем, что они имеют имя. Именованные блоки могут быть разбиты на категории следующим образом:

• Помеченные блоки (labeled block) являются анонимными блоками с меткой, которая дает блоку имя. Они создаются обычно динамически и выполняются только один раз. Помеченные блоки используются так же, как и анонимные блоки, но метка позволяет ссылаться на переменные, которые иначе были бы недоступны.

• Подпрограммы (subprogram) делятся на процедуры и функции. Они могут храниться в базе данных как автономные объекты, как часть пакета или как методы объектного типа. Подпрограммы обычно не меняются после своего создания и выполняются неоднократно.

• Триггеры (triggers) — это именованные блоки, которые ассоциируются с некоторым событием, происходящим в базе данных. Они, как правило, не изменяются после своего создания и выполняются многократно неявным образом при наступлении соответствующих событий. Событием, активизирующим триггер, может быть выполнение оператора языка манипулирования данными (DML, data manipulation language) над некоторой таблицей базы данных. Это может также быть оператор языка определения данных (DDL, data definition language), такой как CREATE или DROP, или событие базы данных, например запуск или остановка.

SQL и PL/SQL



Запросы формируются при помощи SQL, что создает в сети большое число пересылок, по одной на каждый SQL-оператор (см. левую схему на рис.). Однако несколько SQL-операторов можно объединить в один блок PL/SQL и послать их серверу как единое целое (см. правую схему на рис.). В результате сетевой трафик снижается, и приложение функционирует намного быстрее.

Все операторы PL/SQL являются либо процедурными, либо SQL-операторами. (Следует различать два понятия: оператор (operator) — действие, которое может быть выполнено над одним или несколькими операндами для получения результата, и процедурный либо SQL-оператор (statement) — команда (группа команд), используемая для создания программ на языке PL/SQL.) В состав процедурных операторов входят объявления переменных, вызовы процедур и циклические конструкции. SQL-операторы используются для организации доступа к базам данных.

Лексические основы

Любая программа PL/SQL состоит из лексических единиц. Этот набор состоит из следующих символов:

Буквы верхнего и нижнего регистров A-Z и a-z

Цифры 0-9

Разделители: символы табуляции, пробелы и символы возврата каретки

Математические символы +-*/<>=

Символы пунктуации (){}[]?!-;:.’ ” @#%$^&_|

Лексические единицы подразделяются на идентификаторы, ограничители, литералы и комментарии.

Идентификаторы.

Идентификаторы используются для именования объектов PL/SQL, таких как переменные, курсоры, типы и подпрограммы. Идентификатор начинается с буквы, за которой может следовать любая последовательность символов, состоящая из букв, цифр, знаков доллара, знаков подчеркивания и знаков #. Другие символы запрещены. Максимальная длина идентификатора — 30 символов, причем все они являются значащими.

PL/SQL не учитывает регистр символов.

Хорошим стилем программирования считается создание наглядных идентификаторов и корректной схемы именования.

Многие идентификаторы, называемые зарезервированными (или ключевыми) словами, имеют в PL/SQL особое значение. Использовать эти слова для именования собственных идентификаторов нельзя. Например, ключевые слова BEGIN и END применяются для ограничения блоков PL/SQL, поэтому их нельзя использовать в качестве имен переменных.

Если нужно сделать идентификатор чувствительным к регистру символов, включить в его состав такие символы, как пробелы, или воспользоваться зарезервированными словами, следует заключить его в двойные кавычки.

Максимальная длина идентификаторов в кавычках также равна 30 символам (без учета кавычек). В состав идентификатора в кавычках может входить любой печатный символ, за исключением двойных кавычек.

Идентификаторы в кавычках полезны в случае необходимости использовать зарезервированное слово PL/SQL в SQL-операторе. В PL/SQL зарезервировано больше слов, чем в SQL.

Ограничители.

Ограничители — это символы (один символ или их последовательность), которые имеют специальное значение в PL/SQL. Они применяются для отделения идентификаторов друг от друга. Примеры: = - || (конкатенация строк).

Литералы.

Литерал — это символьное, числовое или логическое значение, которое не является идентификатором, например -23.456 и NULL.

Символьные литералы (называемые также строковыми) состоят из одного или нескольких символов, заключенных в одиночные кавычки. (Это такой же стандарт, как и SQL.) Символьные литералы могут быть присвоены переменным, имеющим типы CHAR и VARCHAR2, без преобразования.

Считается, что все строковые литералы имеют тип CHAR. Частью литерала может быть любой печатный символ из набора символов PL/SQL, в том числе и одиночная кавычка. Чтобы включить одиночную кавычку в строковый литерал, нужно расположить две одиночные кавычки рядом друг с другом. Например, превратить строку "Mike's string" в литерал следует так:

'Mike''s string'

Следовательно, в PL/SQL строка, состоящая лишь из одной одиночной кавычки, будет выглядеть следующим образом: ' ' ' '.

Числовой литерал может представлять как целое, так и действительное значение. Числовые литералы могут быть присвоены без преобразования переменным, имеющим тип NUMBER. Это единственный вид литералов, которые разрешено использовать в арифметических выражениях.

Целые литералы состоят из цифр, перед которыми может идти знак + или -. В таких литералах запрещается указывать десятичную точку.

Действительные литералы состоят из цифр (с указанием десятичной точки), перед которыми может быть указан знак + или -.

Существуют только три логических литерала: TRUE (истина), FALSE (ложь) и NULL. Эти значения могут быть присвоены лишь логическим (булевым) переменным.

Комментарии.

Комментарии повышают удобочитаемость программ и делают их более понятными. Компилятор PL/SQL игнорирует комментарии. Существуют комментарии двух видов: однострочные и многострочные. Последние часто называют комментариями С-типа.

Однострочный комментарий начинается с двух символов тире и продолжается до конца строки (ограниченной символом возврата каретки).

Многострочные комментарии начинаются с ограничителя /* и заканчиваются ограничителем */, как это делается в языке программирования С.

Объявления переменных

Переменные определяются в разделе объявлений блока. Каждая переменная имеет конкретный тип, описывающий тип хранящейся в ней информации.

Переменные определяются в разделе объявлений блока. Общий синтаксис объявления переменных:

имя_переменной тип [CONSTANT] [NOT NULL] [:= значение];

где имя_переменной — это имя переменной, тип — это тип, а значение — начальное значение переменной.

Если начальное значение переменной не задано (она не инициализирована), по умолчанию ей присваивается NULL. Если в объявлении указано NOT NULL, переменная должна быть инициализирована. Более того, переменной, которая описана как NOT NULL, запрещается присваивать NULL при ее объявлении, в выполняемом разделе или в разделе исключительных ситуаций блока.

Если в объявлении переменной указано CONSTANT, то она должна быть инициализирована и ее начальное значение не может быть изменено.

При желании вместо := можно воспользоваться ключевым словом DEFAULT (по умолчанию).

Типы данных.

К допустимым **скалярным** типам относятся типы, аналогичные тем, что применяются для определения столбцов таблиц базы данных, плюс ряд дополнительных типов. Скалярные типы можно разделить на семь семейств: числовые типы, символьные типы, типы RAW (для хранения двоичных данных), временные типы, типы ROWID, логические типы и типы Trusted.

Типы данных RAW служат для хранения двоичных данных. При необходимости Oracle автоматически преобразует символьные переменные, для которых применяются разные наборы символов. Это может происходить, когда информация передается из одной базы данных в другую посредством соединения баз данных, каждая из которых использует свой набор символов. Для переменных, имеющих необработанный тип (raw type), это не выполняется.

Тип PL/SQL ROWID абсолютно аналогичен типу, используемому для работы с псевдостолбцами ROWID базы данных. Он дает возможность сохранять идентификаторы строк (rowid), которые можно рассматривать в качестве ключей, однозначно определяющих каждую строку базы данных

В PL/SQL доступны следующие **составные** типы: записи, таблицы (вложенные и индексные) и изменяемые массивы. Составной тип содержит компоненты. В переменной, имеющей составной тип, находится одна или несколько скалярных переменных (называемых также атрибутами).

Ссылочный тип PL/SQL — это то же самое, что и указатель в C. Переменная, объявленная как ссылочная, во время выполнения программы может указывать на различные области памяти.

Типы LOB используются для хранения больших объектов. Большой объект (large object) может быть либо двоичным, либо символьным значением размером до 4 Гбайт.

Объектный тип является составным типом, который имеет внутри себя атрибуты (переменные других типов) и методы (подпрограммы).

Во многих случаях для работы с данными, хранимыми в таблицах базы данных, используются переменные PL/SQL. При этом переменной, работающей с некоторым столбцом, следует присваивать тип, соответствующий типу столбца.

Вместо жесткого задания типа переменной можно воспользоваться атрибутом %TYPE. Он добавляется к ссылке на столбец таблицы или к другой переменной и возвращает ее тип. Например:

DECLARE

v_FirstName students.first_name%TYPE;

При использовании %TYPE переменная v_FirstName будет иметь тот тип, который присвоен столбцу first_name таблицы students. Тип определяется всякий раз, когда данный блок

выполняется для анонимных и именованных блоков и когда компилируются хранимые объекты (процедуры, функции и т.д.).

Подтипы, определяемые пользователем

Подтип (subtype) — это тип PL/SQL, в основе которого лежит существующий тип. С помощью подтипа можно дать типу альтернативное имя, которое более точно описывает его назначение. Ряд подтипов PL/SQL (например, DECIMAL и INTEGER - подтипы NUMBER) определен в модуле STANDARD. Для определения подтипа используется следующий синтаксис:

SUBTYPE новый_тип IS исходный_тип;

где новый_тип — имя нового подтипа, а исходный_тип указывает базовый тип. Базовый тип может быть либо предопределенным типом, либо подтипом, либо ссылкой %TYPE.

Преобразование типов данных

PL/SQL может выполнять преобразования между различными семействами скалярных типов данных.

Существуют два способа преобразования типов данных: явный и неявный.

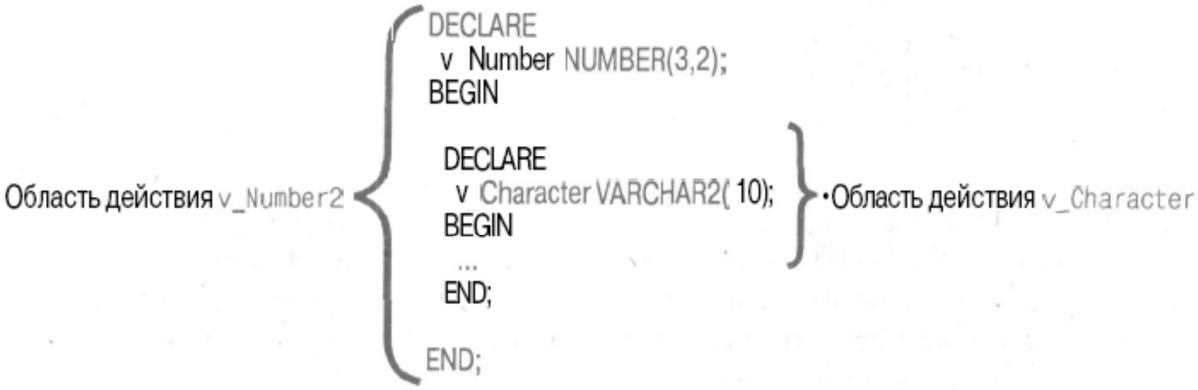
Явное преобразование типов данных. Встроенные функции преобразования, доступные в SQL, доступны и в PL/SQL. Например, TO_DATE, TO_CHAR и т.д.

Неявное преобразование типов данных. В PL/SQL осуществляется автоматическое преобразование типов данных разных семейств, когда это возможно. Хотя в PL/SQL производится неявное преобразование типов данных, при программировании рекомендуется использовать явные функции преобразования.

Области действия и видимости

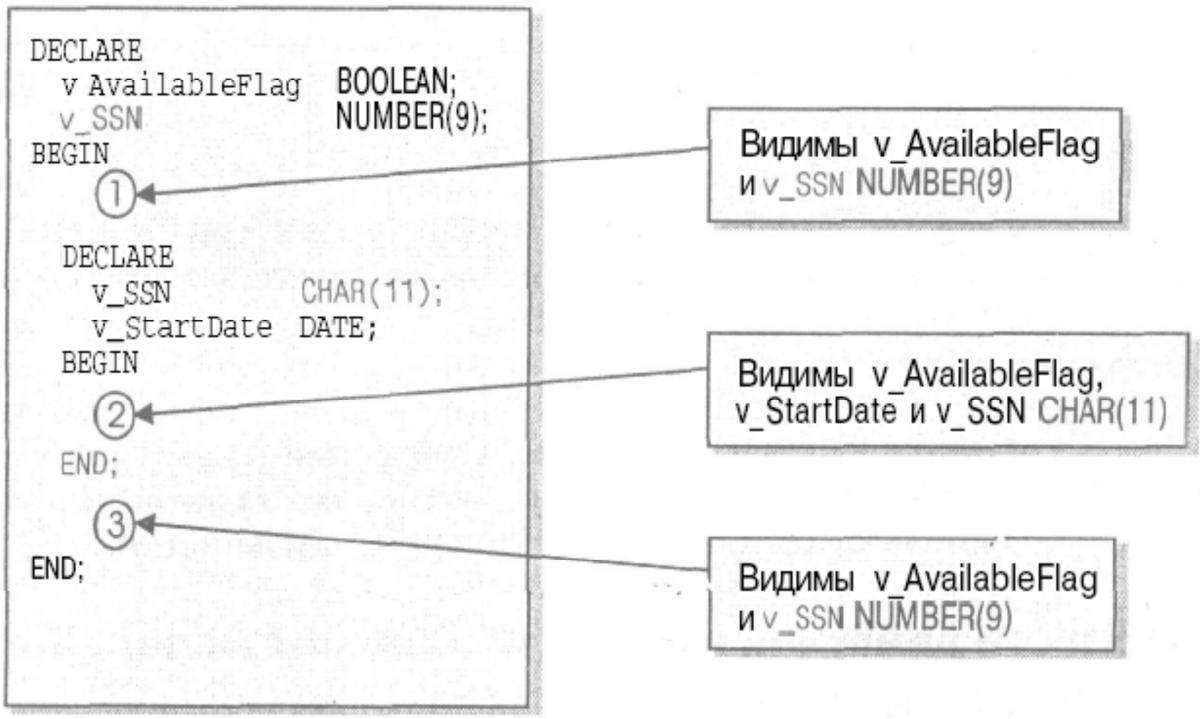
Область действия (scope) переменной — это фрагмент программы, в котором возможно обращение к данной переменной. Для переменной PL/SQL это фрагмент с момента ее объявления до конца блока. Когда переменная выходит из своей области действия, PL/SQL освобождает память, используемую для хранения переменной, и ссылки на нее становятся невозможны.

На рис. показано, что областью действия переменной v_Character является только внутренний блок; после ключевого слова END внутреннего блока она становится недоступной. Область действия переменной v_Number распространяется до ключевого слова END внешнего блока. Внутренний блок является областью действия обеих переменных.



Область видимости (visibility) переменной — это фрагмент программы, в котором возможно обращение к этой переменной без использования ее квалифицированного имени. Область видимости всегда лежит в пределах области действия; если переменная находится вне области своего действия, она невидима.

Рассмотрим рис. В точке 1 переменные v_AvailableFlag и v_SSN находятся в границах своих областей действия и видимы. В точке 2 обе эти переменные находятся в пределах своих областей действия, но видима только переменная v_AvailableFlag. При объявлении переменной v_SSN типа CHAR(11) объявление NUMBER(9) становится скрытым.



Достучаться до v_SSN типа NUMBER(9) из внутреннего блока можно с помощью метки. Для этого перед Declare внешнего блока необходимо создать метку, например, «1_Outer». Это делает внешний блок помеченным. При обращении из точки 2 использовать синтаксис: 1_Outer.v_SSN. Более подробно метки будут разбираться в одной из следующих глав.

Выражения и операции

Посредством выражений и операций осуществляется связывание переменных PL/SQL. С помощью операций определяются способы присваивания переменным конкретных значений и способы работы с этими значениями. Выражение (expression) — это некоторая последовательность переменных и литералов, разделенных знаками операций, или операторами (operator). Значение выражения определяется значениями переменных и литералов, составляющих это выражение, а также описанием используемых операций.

Присваивание

Основной операцией является операция присваивания (assignment). Ее синтаксис таков:

переменная := выражения;

где переменная — это переменная PL/SQL, а выражение — это выражение PL/SQL.

В отличие от других языков программирования, например C, в любом операторе (statement) PL/SQL может быть только одна операция присваивания.

Выражения

Выражения PL/SQL используются в качестве значений выражения (rvalues), поэтому использование выражения как самостоятельного оператора бессмысленно — оно должно быть элементом оператора. Тип выражения определяют составляющие его операции и типы операндов.

Арифметические выражения

Используются стандартные правила для арифметических выражений

Символьные выражения

Существует лишь одна символьная операция — операция конкатенации (||). С ее помощью соединяются две или большее количество символьных строк (или аргументов, которые могут быть неявно преобразованы в символьные строки).

Логические выражения

Во всех управляющих структурах PL/SQL (за исключением GOTO) используются логические выражения, называемые также условиями. Логическое, или булево, выражение — это любое выражение, которое дает в результате логическое значение (TRUE (истина), FALSE (ложь) или NULL). Ниже приведен ряд логических выражений:

X > Y

NULL

(4 > 5) OR (-1 != Z)

В трех логических операциях — AND, OR и NOT — логические значения используются в качестве аргументов и возвращаются в качестве результата.

В операциях сравнения (>, <, >=, <=, =, !=), или отношения, в качестве операндов используются числа, символы или данные, а возвращаются логические значения (операция сравнения «не равно» может быть следующей: != либо <>).

Оператор IS NULL возвращает значение TRUE только тогда, когда операндом является NULL. NULL-значения не могут быть проверены на истинность при помощи операций отношения, так как любое выражение отношения, операндом которого является NULL, возвращает NULL.

Оператор LIKE (подобие) применяется для сопоставления строк символов с некоторым образцом. Знак подчеркивания (_) соответствует одному символу, а знак процента (%) — нулю и более символам.

Оператор BETWEEN (между) объединяет операции <= и >= в одном выражении. Например, приведенное ниже выражение возвращает значение FALSE:

100 BETWEEN 110 AND 120

Оператор IN (в) возвращает TRUE, если первый операнд содержится в наборе, определяемом вторым операндом. Например, результат этого выражения — FALSE:

'Scott' IN ('Mike', 'Pamela', 'Fred')

Управляющие структуры PL/SQL

В PL/SQL, как и в других языках программирования третьего поколения, имеются различные структуры, служащие для управления работой блока.

Этими структурами являются условные операторы и циклы. Именно эти структуры совместно с переменными обеспечивают мощь и гибкость PL/SQL.

IF-THEN-ELSE

Синтаксис оператора IF-THEN-ELSE:

IF логическое_выражение1 THEN
последовательъностъ_операторов1;
[ELSIF логическое_выражение2 THEN
последовательъностъ_операторов2;]
…
[ELSE
последовательностъ_операторов3;]
END IF;

где логическое_выражение — любое выражение, результатом которого является логическое значение. Условия ELSIF и ELSE необязательны, причем условий ELSIF может быть сколь угодно много.

В последовательностях операторов может содержаться несколько операторов, разделенных ;.

Последовательность операторов в операторе IF-THEN-ELSE выполняется только в случае истинности соответствующего условия. Если результатом условия является FALSE или NULL, последовательность операторов не выполняется.

В качестве примера рассмотрим два блока:

```
/* Блок 1 */
DECLARE
v_Number1 NUMBER;
v_Number2 NUMBER;
v_Result VARCHAR2(7);
BEGIN
  IF v_Number1 < v_Number2 THEN
        v_Result := 'Yes';
  ELSE
        v_Result := 'No';
  END IF;
END;

/* Блок 2 */
DECLARE
v_Number1 NUMBER;
v_Number2 NUMBER;
v_Result VARCHAR2(7);
BEGIN
  IF v_Number1 >= v_Number2 THEN
        v_Result := 'No';
  ELSE
        v_Result ;= 'Yes';
  END IF;
END;
```

Предположим, что v_Number1 = 3, а v_Number2 = 7. В результате условие блока 1 (3 < 7) будет истинным, и переменная v_Result будет установлена в 'Yes'. Условие блока 2 будет ложным (3 >= 7), переменная v_Result также будет установлена в 'Yes'. Для любых значений v_Number1 и v_Number2, не являющихся NULL, приведенные блоки функционируют одинаково.

Теперь предположим, что v_Number1 = 3, а v_Number2 содержит NULL.

Что произойдет в этом случае? Условие блока 1 (3 < NULL) дает в результате NULL, поэтому будет выполнено условие ELSE и переменной v_Result будет присвоено значение 'No'. Условие блока 2 (3 >= NULL) также дает в результате NULL, поэтому будет выполнено условие ELSE и переменной v_Result будет присвоено значение 'Yes'. Если одна из переменных (v_Numberl или v_Number2) содержит NULL, то блоки функционируют по-разному.

Можно сделать так, чтобы блоки функционировали одинаково. Для этого следует ввести проверку на наличие NULL:

```
/* Блок 1 */
DECLARE
v_Number1 NUMBER;
v_Number2 NUMBER;
v_Result VARCHAR2(7);
BEGIN
  IF v_Number1 IS NULL OR
  v_Number2 IS NULL THEN
        v_Result := 'Unknown’;
  ELSIF v_Number1 < v_Number2 THEN
        v_Result := ' Y e s ' ;
  ELSE
        v_Result := ' No ' ;
  END IF;
```

```
END;

/* Блок 2 */
DECLARE
v_Number1 NUMBER;
v_Number2 NUMBER;
v_Result VARCHAR2(7);
BEGIN
  IF v_Numberl IS NULL OR
  v_Number2 IS NULL THEN
        v_Result := 'Unknown'
  ELSIF v_Number1 >= v_Number2 THEN
        v_Result := 'No';
  ELSE
        v_Result := 'Yes';
  END IF;
END;
```

Условие IS NULL истинно только тогда, когда проверяемая переменная содержит NULL-значение. В противном случае условие ложно. После добавления в блоки проверки на наличие NULL переменная v_Result будет принимать значение 'Unknown' (неизвестно), если одна из переменных v_Number содержит NULL- значение. Сравнение переменных v_Number1 и v_Number2 будет выполняться только в том случае, если точно известно, что обе переменные не содержат NULL- значений; при этом блоки будут функционировать одинаково.

CASE

Аналогично оператору переключателя в C, CASE имеет следующую структуру:

```
CASE test_var
WHEN значение_1 THEN последовательъностъ_операторов_1;
WHEN значение_2 THEN последовательностъ_операторов_2;
…
WHEN значение_n THEN последовательность_операторов_n;
[ELSE последовательность_else;]
END CASE;
```

где test_var— проверяемая переменная или выражение, значение_1 … значение_n — сравниваемые значения, а последовательъностъ_операторов_1 … последовательъностъ_операторов_n — соответствующий код для выполнения. Если test_var равна, например, значению_2, будет выполняться последовательъностъ_операторов_2. Если ни одно из значений не совпадает, выполняется код последовательъностъ_else.

Элемент test_var, используемый в операторе CASE, может быть не только переменной, как в примере выше. Это может быть выражение произвольной сложности, содержащее вызовы функций. В любом случае test_var оценивается только один раз, в начале выполнения оператора CASE. Более того, типы значений_1... n должны быть совместимые типом test_var.

После выполнения заданной последовательности операторов управление немедленно передается оператору, следующему за CASE. Не требуется использовать оператор break, как в случае переключателя C.

Предложение ELSE оператора CASE является необязательным. В случае если оно не указано, а проверяемое выражение не соответствует ни одному из сравниваемых значений, PL/SQL порождает предопределенную ошибку CASE_NOT_FOUND, которая эквивалента ORA-6592

Операторы CASE с поиском

В рассмотренных выше примерах использовались операторы CASE с проверкой — каждое предложение WHEN сравнивало значение с единственным тестовым выражением. Применяются также операторы CASE с поиском, они имеют следующую структуру:

```
CASE
  WHEN testl THEN последовательность_операторов_1;
  WHEN test2 THEN последовательность_операторов_2;
  …
  WHEN testn THEN последовательность_операторов_n;
  [ELSE последовательность_else;]
END CASE;
```

Здесь нет проверяемого выражения; вместо этого каждое предложение WHEN содержит логическое выражение. Подобно оператору IF-THEN, если тестовое выражение оценивается как TRUE, то выполняется соответствующая последовательность операторов. Следующий пример иллюстрирует использование оператора CASE с поиском. Отметим, что тестовые выражения не обязаны проверять один и тот же элемент.

```
DECLARE
v_Test1 NUMBER := 2;
v_Test2 VARCHAR2(20) := 'Goodbye';
BEGIN
  CASE
        WHEN v_Test1 = 1 THEN
              DBMS_OUTPUT.PUT_LINE('One! ');
              DBMS_OUTPUT.PUT_LINE('Another one! ');
        WHEN v_Test1 > 1 THEN
              DBMS_OUTPUT.PUT_LINE('> 1!');
              DBMS_OUTPUT.PUT_LINE('Still > 1! ');
        WHEN v_Test2 = 'Goodbye!' THEN
              DBMS_OUTPUT.PUT_LINE(‘Goodbye! ');
              DBMS_OUTPUT.PUT_LINE('Adios! ');
        ELSE
              DBMS_OUTPUT.PUT_LINE('No match');
  END CASE;
END;
```

В этом примере показаны предложения WHEN, имеющие в своем составе более одного оператора. Отметим также: хотя v_Test2 равно 'Goodbye' и, следовательно, третье условие WHEN будет истинным, оно не выполняется, так как предыдущее условие уже было оценено как TRUE.

Циклы

В PL/SQL можно повторять операторы посредством циклов. Циклы подразделяются на четыре категории. Простые циклы, циклы WHILE и циклы FOR рассматриваются ниже. Курсорные циклы FOR будут рассмотрены в следующих лекциях.

Простые циклы

Синтаксис простых циклов (основных циклов языка) таков:

```
LOOP
  последовательностъ_операторов;
END LOOP;
```

Последовательность операторов будет выполняться бесконечно долго, так как в этом цикле отсутствует условие его завершения. Такое условие можно предусмотреть, если добавить оператор EXIT (выход), имеющий следующий синтаксис:

```
EXIT [WHEN условие]
```

Рассмотрим блок, с помощью которого в таблицу temp_table вводится 50 строк:

```
DECLARE
v_Counter BINARY_INTEGER := 1;
BEGIN
  LOOP
        INSERT INTO temp_table
        VALUES (v_Counter, 'LOOP index');
        v_Counter := v_Counter + 1;
        IF v_Counter > 50 THEN EXIT;
        END IF;
  END LOOP;
END;
```

Оператор

```
EXIT WHEN условие
```

эквивалентен оператору

```
IF условие THEN
  EXIT;
END IF;
```

Поэтому можно изменить приведенный блок, но его функционирование останется прежним:

```
DECLARE
v_Counter BINARY_INTEGER := 1;
BEGIN
  LOOP
        INSERT INTO temp_table
        VALUES (v_Counter, 'Loop index');
        v_Counter := v_Counter + 1;
        EXIT WHEN v_Counter > 50;
  END LOOP;
END;
```

Циклы WHILE

Синтаксис цикла WHILE (цикл с условием продолжения) таков:

```
WHILE условие LOOP
  последовательностъ_ операторов;
END LOOP;
```

Проверка условия происходит перед каждой итерацией (шагом) цикла. Если условие истинно, выполняется последовательность операторов. Если же проверка условия дает ложное или NULL-значение, цикл завершается, и управление программой передается оператору, следующему за оператором END LOOP. Теперь перепишем рассматриваемый блок, применив цикл WHILE:

```
DECLARE
v_Counter BINARY_INTEGER := 1;
BEGIN
  WHILE v_Counter <= 50 LOOP
        INSERT INTO temp_table
        VALUES (v_Counter, 'Loop index');
        v_Counter := v_Counter + 1;
  END LOOP;
END;
```

Чтобы прервать цикл и выйти из него, можно внутри цикла WHILE использовать оператор EXIT или EXIT WHEN.

Числовые циклы FOR

Число итераций в простых циклах и циклах WHILE не известно заранее — оно зависит от условий, заданных в циклах. В числовых же циклах FOR число итераций заранее определено. Синтаксис цикла FOR таков:

```
FOR счетчик_цикла IN [REVERSE] нижняя_граница .. верхняя_граница LOOP
  последовательность_ операторов;
END LOOP;
```

где счетчик_цикла — неявно создаваемая индексная переменная, нижняя_граница и верхняя_граница указывают число итераций, а последовательъностъ_операторов — это содержимое цикла.

Границы цикла вычисляются один раз и определяют общее число итераций, проходимых счетчиком цикла от нижней границы до верхней границы. При этом счетчик каждый раз увеличивается на 1 до тех пор, пока цикл не завершится. Представим наш пример с помощью цикла FOR:

```
BEGIN
  FOR v_Counter IN 1..50 LOOP
        INSERT INTO temp_table
        VALUES (v_Counter, 'Loop Index');
  END LOOP;
END;
```

Счетчик (индекс) цикла FOR неявно объявляется как BINARY_INTEGER. Объявлять его перед циклом необязательно. Если он все же объявлен, индекс цикла скрывает внешнее объявление так же, как объявление переменной во внутреннем блоке скрывает ее объявление во внешнем блоке:

Если в цикле FOR указывается ключевое слово REVERSE (обратный порядок), индекс цикла будет изменяться от верхней границы до нижней. Обратите внимание, что в этом случае синтаксис остается прежним — нижняя граница по-прежнему указывается первой.

Диапазоны цикла Верхняя и нижняя границы необязательно должны быть числовыми литералами. Они могут задаваться любыми выражениями, для которых возможно преобразование в числовые значения. Это позволяет динамически определять верхнюю и нижнюю границу.

Операторы GOTO и метки

В языке программирования PL/SQL имеется оператор GOTO. Его синтаксис:

```
GOTO метка,
```

где метка — это метка, определяемая в блоке PL/SQL. Метки заключаются в двойные угловые скобки. При выполнении оператора GOTO управление программой сразу же передастся оператору, на который указывает метка. Представим рассматриваемый пример цикла следующим образом:

```
DECLARE
v_Counter BINARY_INTEGER := 1;
BEGIN
  LOOP
        INSERT INTO temp_table
        VALUES (v_Counter, 'Loop count');
        v_Counter := v_Counter + 1;
        IF v_Counter > 50 THEN
              GOTO l_EndOfLoop;
        END IF;
  END LOOP;
<<l_EndOfLoop>>
INSERT INTO temp_table (char_col)
VALUES ('Done!');
```

```
END;
```

В PL/SQL на использование операторов GOTO налагаются определенные ограничения. Нельзя передавать управление программой во внутренний блок, цикл или оператор IF. Кроме того, запрещается передавать управление из одной последовательности операторов условного оператора IF в другую. Наконец, нельзя передавать управление из обработчика исключительных ситуаций обратно в текущий блок.

Рекомендации по применению GOTO

При использовании операторов GOTO следует соблюдать осторожность. Лишние операторы GOTO могут привести к нарушению структуры программы: управление программой может перемещаться с места на место без видимой причины, что затруднит ее понимание и сопровождение. Все случаи применения операторов GOTO могут быть представлены с помощью других управляющих структур PL/SQL, например циклов или условных выражений. Для выхода из вложенного цикла можно вместо оператора перехода к концу блока воспользоваться исключительной ситуацией.

Помеченные циклы

Циклы могут быть помечены. При этом метка может быть указана в операторе EXIT для определения цикла, который нужно прервать. Например:

```
BEGIN
«l_Outer»
FOR v_Outerlndex IN 1.. 50 LOOP
  …
  «l_Inner»
  FOR v_lnnerIndex IN 2.. 10 LOOP
  …
        IF v_OuterIndex > 40 THEN
              EXIT l_Outer; - выход из обоих циклов
        END IF;
  END LOOP l_Inner;
END LOOP l_Outer;
END;
```

Если цикл помечен, имя метки можно указывать после оператора END LOOP, как показано в этом примере.

NULL как оператор

В некоторых случаях необходимо явно указывать, что никаких действий выполнять не нужно. Это можно сделать при помощи оператора NULL, который не приводит ни к каким действиям; он служит лишь в качестве заглушки.

Синтаксис: Null;

Записи PL/SQL

Скалярные типы (NUMBER, VARCHAR2, DATE и т.д.) предопределены в модуле STANDARD. Поэтому для использования скалярного типа в программе нужно лишь объявить переменную этого типа. С другой стороны, составные типы определяются пользователем. Для применения составного типа необходимо сначала определить тип, а затем объявить переменные этого типа — аналогично синтаксису объявления подтипа.

Записи PL/SQL аналогичны структурам С. Запись предоставляет способ работы с отдельными, но связанными между собой переменными как с целым.

Общий синтаксис определения типа записи:

```
TYPE record_type IS RECORD (
  fieldl typel [NOT NULL] [:= exprl],
  field2 type2 [NOT NULL] [:= expr2],
```

…
fieldn typen [NOT NULL] [:= exprn]),

где record_type — имя нового типа, поля field1 ... fieldn — имена полей записи, а типы type1 ... typen — типы соответствующих полей. Запись может иметь сколько угодно полей. Объявление каждого поля выглядит по сути так же, как объявление переменной вне записи, включая ограничения NOT NULL и начальные значения.

Обратиться к полю внутри записи можно следующим образом:

имя_записи.имя_поля

Присваивание записей

Чтобы присвоить одну запись другой, обе записи должны быть одного типа. v_Sample1 := v_Sample2;

Такое присваивание записей использует семантику копирования — значениям полей в v_Sample1 будут присваиваться значения соответствующих полей в v_Sample2. В случае же двух различных типов, имеющих одинаковые определения полей, записи нельзя присвоить друг другу. В этом случае возникает ошибка "PLS-382: expression is of wrong type" (выражение неверного типа).

Запись можно указывать в операторе SELECT. При этом данные из базы данных будут извлекаться и помещаться в запись. Поля записи должны соответствовать полям, выбираемым в запросе. Следующий пример иллюстрирует такое использование записей:

```
DECLARE
TYPE t_StudentRecord IS RECORD (
  FirstName students.first_name%TYPE;
  LastName students.last_name%TYPE;
  Major students.major%TYPE);
v_Student t_StudentRecord;
BEGIN
  SELECT first_name, last_name, major
  INTO v_Student
  FROM students
  WHERE ID = 10000;
END;
```

Использование %ROWTYPE

Довольно часто в PL/SQL для объявления записи применяются типы данных, соответствующие строке базы данных. Для этих целей PL/SQL предоставляет оператор %ROWTYPE. Аналогично %TYPE, %ROWTYPE возвращает тип данных на основе определения таблицы. Например, такое объявление, как

```
DECLARE
    v_RoomRecord rooms%ROWTYPE;
```

определяет запись, поля которой соответствуют столбцам таблицы rooms.

Как и в случае %TYPE, сюда не включается ограничение NOT NULL, определенное для столбца. Однако длина столбцов VARCHAR2 и CHAR, а также точность и масштаб для столбцов NUMBER учитываются.

Если определение таблицы изменяется, то %ROWTYPE изменяется вместе с ним. Подобно %TYPE, %ROWTYPE определяется всякий раз, когда анонимный блок передается среде выполнения PL/SQL, и всякий раз, когда компилируется хранимый объект.

SQL в PL/SQL

Как мы говорили, язык SQL состоит из двух основных классов: DML (SELECT, INSERT, UPDATE, DELETE, EXPLAIN PLAN) и DDL (DROP, CREATE, ALTER, GRANT, REVOKE). Несколько особняком стоят группы операторов по управлению транзакциями, сеансом работы (ALTER SESSION, SET ROLE) и системой (ALTER SYSTEM).

Из всех SQL-операторов в программах PL/SQL можно применять лишь операторы DML и операторы управления транзакциями. Операторы DDL использовать нельзя. Это обусловлено тем, что в PL/SQL используется раннее связывание (binding) переменных с соответствующими областями памяти. В PL/SQL в процесс привязки входит также проверка базы данных на наличие полномочий, позволяющих обращаться к объектам схем. В том языке, где используется ранняя привязка (early binding), этот процесс осуществляется на этапе компиляции программы, а в языке, где применяется поздняя привязка (late binding), она откладывается до этапа выполнения программы. Ранняя привязка означает, что компиляция программы будет занимать большее время (так как при этом нужно привязывать переменные), однако выполняться такая программа будет быстрее, потому что привязка уже завершена. Поздняя привязка сокращает время компиляции, но увеличивает время выполнения программы.

Тем не менее, существует способ, обеспечивающий выполнение в PL/SQL всех допустимых операторов SQL, включая DDL. Это динамический SQL. Он позволяет создавать оператор SQL динамически, во время выполнения программы, а затем проводить его синтаксический анализ и выполнение.

Execute Immediate ‘строка SQL-запроса’

Переменные могут указываться в любом месте SQL-оператора, где разрешены выражения. Переменные, используемые таким образом, называются переменными присваивания или переменными привязки (bind variable).

В SQL-операторе разрешается заменять переменными только выражения. Особенно отметим, что имена таблиц и столбцов должны быть известны. Дело в том, что при ранней привязке имена объектов Oracle должны быть известны во время компиляции. По определению, значение переменной до выполнения программы неизвестно. Чтобы обойти это ограничение, можно вновь воспользоваться динамическим SQL.

Динамический SQL

Если набор команд для работы с данными в программе довольно разнообразен, было бы непрактично заранее предопределять и внедрять в программу все необходимые выражения SQL, соответствующие всем возможным командам. Вместо этого, вероятно, целесообразнее конструировать необходимые запросы SQL динамически, а затем динамически же компилировать и выполнять сконструированные запросы.

Для поддержки данного процесса предназначены средства динамического языка SQL, описанные в этом разделе. Динамический SQL разбирается и исполняется во время выполнения, а не синтаксического разбора блока PL/SQL.

Существуют два способа выполнения динамического SQL в PL/SQL.

Первый применяет модуль DBMS_SQL. Второй способ предлагает использование встроенного динамического SQL. Встроенный динамический SQL является составной частью самого языка. Вследствие этого он значительно проще в применении и быстрее, чем модуль DBMS_SQL.

Блоки PL/SQL и выполнение операторов, не содержащих запросов

Базовым оператором, используемым в не содержащих запросов операторах (DML и DDL) и блоках PL/SQL, является оператор EXECUTE IMMEDIATE.

Синтаксис:

EXECUTE IMMEDIATE строка [USING var1 [, var2]…]

Пример.

```
DECLARE
v_SQLString VARCHAR2(200);
v_PLSQLBlock VARCHAR2(200);
BEGIN
EXECUTE IMMEDIATE
'CREATE TABLE execute_table (call VARCHAR2(10))';
FOR v_Counter IN 1..10 LOOP
```

```
v_SQLString :=
'INSERT INTO execute_table
VALUES (''Row' || v_Counter || ''')';
EXECUTE IMMEDIATE v_SQLString;
END LOOP;
v_PLSQLBlock :=
'BEGIN
FOR v_Rec IN (SELECT * FROM execute_table) LOOP
DBMS_OUTPUT.PUT_LINE(v_Rec.call);
END LOOP;
END;';
EXECUTE IMMEDIATE v_PLSQLBlock;
EXECUTE IMMEDIATE 'DROP TABLE execute_table';
END;
```

В этом примере показаны различные способы использования EXECUTE IMMEDIATE: для выполнения DDL, DML и анонимных блоков PL/SQL.

Выполняемая строка может задаваться как литерал, заключенный в одиночные кавычки (операторы CREATE TABLE и DROP TABLE) или как переменная типа символьной строки PL/SQL (оператор INSERT и анонимные блоки). Отметим, что завершающая точка с запятой не нужна для операторов DML и DDL, но указывается для анонимных блоков.

EXECUTE IMMEDIATE используется также для выполнения операторов со связанными переменными. В этом случае выполняемая строка содержит специальные позиции, помеченные двоеточием. Позиции предназначены для размещения переменных PL/SQL, которые указываются в предложении USING оператора EXECUTE IMMEDIATE, например:

```
DECLARE
v_SQLString VARCHAR2(1000);
v_PLSQLBlock VARCHAR2(1000);
CURSOR c_EconMajor IS
SELECT *
FROM students
WHERE major = 'Economies';
BEGIN
v_SQLString :=
'INSERT INTO CLASSES (department, course, description, max_students, current_students, num_credits)
VALUES(:dep, :course, :descr, :max_s, :cur_s, :num_c)';
EXECUTE IMMEDIATE v_SQLString USING
'ECN', 103, 'Economics 103', 10, 0, 3;
FOR v_StudentRec IN c_EconMajor LOOP
ECECUTE IMMEDIATE
'INSERT INTO registered_students
(student_ID, department, course, grade)
VALUES (:id, :dep, :course, NULL)'
USING v_Studentflec.ID, 'ENC', 103;
v_PLSQLBlock :=
'BEGIN
UPDATE classes SET current_students = current_students + 1
WHERE department = :d and course = :c;
END; ' ;
EXECUTE IMMEDIATE v_PLSQLBlock USING 'ECN', 103;
END LOOP;
END;
```

Выполнение запросов

Запросы выполняются с помощью оператора OPEN FOR аналогично курсорным переменным. Различие состоит в том, что строка, содержащая запрос, может быть переменной PL/SQL, а не литералом. К получаемой курсорной переменной можно обращаться так же, как и к любой другой переменной. Для связывания используется предложение USING, так же как в операторе EXECUTE IMMEDIATE.

OPEN v_ReturnCursor FOR v_SQLStatement USING p_Major;

Интерфейсы уровня вызовов SQL/CLI

В этом разделе коснемся интерфейса SQL/CLI, предоставляющего возможности динамического SQL для базовых языков программирования (т.е. процедурных языков, которые используют SQL-запросы и, таким образом, являются базовыми по отношению к языку SQL).

Средства интерфейса уровня вызовов SQL (SQL Call-Level Interface — SQL/CLI, или сокращенно CLI) были введены в стандарт SQL в 1995 году. Интерфейс CLI в основном создан на базе интерфейса ODBC (Open Database Connectivity) компании Microsoft. И тот, и другой интерфейс позволяет приложениям, которые написаны на одном из базовых языков, выдавать запросы к базе данных, обращаясь к процедурам CLI, предоставляемым изготовителем, не прибегая к помощи внедренных операторов SQL. Затем в этих процедурах, которые предварительно должны быть обязательно связаны с данным приложением, используется динамический язык SQL для выполнения требуемых операций с базой данных от имени приложения.

Интерфейс SQL/CLI (а также ODBC) решает ту же задачу, что и динамический язык SQL, а именно — позволяет приложению передавать на выполнение текст оператора SQL именно к тому времени, когда его непосредственно необходимо выполнять. Однако применяемый в интерфейсах CLI и ODBC подход к решению этой задачи организован лучше, чем в динамическом языке SQL. Его преимущества заключаются в следующем.

- Во-первых, динамический язык SQL — это исходный код, который должен соответствовать стандарту SQL. Поэтому для любого приложения, которое использует динамический язык SQL (в частности, для приложения на PL/SQL), требуется какой-то компилятор SQL, необходимый для обработки установленных стандартом операций, таких как EXECUTE IMMEDIATE т.д. Интерфейсом CLI, напротив, нормированы лишь подробности вызова процедур (т.е. в основном вызова подпрограмм). Не требуются средства специального компилятора; достаточно использовать обычный компилятор стандартного базового языка. Поэтому приложение может распространяться (возможно даже сторонними изготовителями программного обеспечения) в "сжатой" форме, в виде объектного кода.

- Во-вторых, такие приложения могут быть независимыми от типа СУБД, т.е. интерфейс SQL/CLI включает средства создания универсальных приложений (опять же, возможно, сторонними изготовителями программного обеспечения), которые могут использоваться для нескольких различных типов СУБД, без специализации с учетом какой-то конкретной СУБД.

Рассмотрим пример, иллюстрирующий использование интерфейса SQL/CLI.

```
char sqlsource [65000];
strcpy ( sqlsource,
"DELETE FROM SP WHERE QTY < QTY ( 300 )" ) ;
rc = SQLExecDirect ( hstmt, (SQLCHAR *)sqlsource,
SQL_NTS ) ;
```

Пояснение

1. Поскольку в реальных приложениях SQL/CLI базовым языком обычно служит C, в данном примере вместо PL/I используется C. Необходимо также учитывать, что интерфейс SQL/CLI представляет собой набор стандартных средств для вызова подпрограмм из базового языка, поэтому его синтаксис (а также, безусловно, соответствующая семантика) изменяется в зависимости от базового языка.

2. Функция strcpy языка C вызывается для копирования исходной формы определенного оператора DELETE языка SQL в переменную sqlsource языка C.

3. Оператор присваивания C ("=") вызывает процедуру SQLExecDirect интерфейса SQL/CLI (аналог оператора EXECUTE IMMEDIATE динамического языка SQL) для выполнения оператора SQL, содержащегося в переменной sqlsource, и присваивает переменной гс программы C код возврата, полученный в результате этого вызова.

Как и можно было бы предположить, интерфейс SQL/CLI в той или иной степени включает аналоги всех средств динамического выполнения SQL, наряду с некоторыми дополнениями. Такие интерфейсы, как CLI, ODBC и JDBC (фактически вариант ODBC для языка Java), приобретают все более важное практическое значение.

Курсоры

В предыдущей теме уже говорилось о применении SQL-операторов в PL/SQL. Курсоры расширяют эти функциональные возможности и позволяют программам явным образом управлять процессом обработки SQL-операторов.

Операторы DML в общем случае работают не с одной, а с множеством строк, в то время как процедурные языки обычно не приспособлены для выборки больше одной строки за одно обращение. Следовательно, необходим своего рода "мост" между предусмотренными в языке SQL средствами манипулирования данными, позволяющими работать одновременно с множеством строк, и применяемыми в процедурном языке средствами обработки, допускающими одновременное использование только одной строки. В качестве подобного "моста" используются курсоры. Курсор представляет собой своего рода логический указатель, который может использоваться в приложении для перемещения по набору строк, указывая поочередно на каждую из них и таким образом обеспечивая возможность адресации этих строк — по одной за один раз.

В этой теме поясняется, как использовать курсоры для многострочных запросов и других SQL-операторов. Кроме того, рассматриваются курсорные переменные, которые обеспечивают более динамичное применение курсоров.

Определение курсора

Для обработки SQL-оператора Oracle выделяет область памяти, называемую контекстной областью (context area). Она содержит информацию, необходимую для завершения обработки, включая: число строк, обрабатываемых оператором, указатель на представление этого оператора после проведения синтаксического анализа и активный набор (active set), т.е. набор строк, возвращаемых запросом.

Курсор (cursor) — это указатель на контекстную область. С его помощью программа PL/SQL может управлять контекстной областью и ее состоянием во время обработки оператора. Ниже приводится блок PL/SQL, в котором выполняется цикл выборки курсора. Здесь запрос возвращает несколько строк данных.

```
DECLARE
/* Выходные переменные для хранения результатов запроса */
v_StudentID students.id%TYPE;
v_FirstName students.first_name%TYPE;
v_LastName students.last_name%TYPE;
/* Переменная привязки, используемая в запросе */
v_Major students.major%TYPE := 'Computer Science';
/* Объявление курсора */
CURSOR c_Students IS
SELECT id, first_name, last_name
FROM students
WHERE major = v_Major;
BEGIN
/* Идентифицируем строки активного набора и подготовимся к дальнейшей обработке данных */
OPEN c_Students;
```

```
LOOP
/* Извлечем каждую строку активного набора в переменные PL/SQL */
FETCH c_Students INTO v_StudentID, v_FirstName, v_LastName;
/* Если строки, которые нужно извлечь, закончились, выходим из цикла */
EXIT WHEN c_Students%NOTFOUND;
END LOOP;
/* Освободим ресурсы, используемые запросом */
CLOSE c_Students;
END;
```

В этом примере используется явный (explicit) курсор. Имя курсора явно присваивается оператору SELECT при помощи оператора CURSOR...IS. Для всех других SQL-операторов применяются неявные (implicit) курсоры.

Обработка явного курсора выполняется в четыре этапа. Обработка неявного курсора осуществляется в PL/SQL автоматически.

Обработка явных курсоров

Для обработки явного курсора в PL/SQL необходимо выполнить четыре действия:

1. Объявить курсор.
2. Открыть курсор для запроса.
3. Выбрать результаты в переменные PL/SQL.
4. Закрыть курсор.

Объявление курсора

Объявление курсора является единственным действием, которое выполняется в разделе объявлений блока. Другие три действия производятся в выполняемом разделе или в разделе исключительных ситуаций.

При объявлении курсора ему назначается имя и ставится в соответствие некоторый оператор SELECT, возвращающий табличное выражение. Табличное выражение не вычисляется при объявлении курсора.

Синтаксис объявления курсора:

CURSOR имя_курсора IS оператор_SELECT

где имя_курсора — это имя курсора, а оператор_SELECT — запрос, который будет обрабатываться. Что касается области действия и области видимости, то курсоры отвечают стандартным правилам, установленным для идентификаторов PL/SQL. Так как имя курсора является идентификатором PL/SQL, оно должно быть объявлено до того, как на него будет произведена ссылка. Можно использовать любые операторы SELECT, в том числе соединения (join) и операторы, в состав которых входят конструкции UNION (объединение) и MINUS (минус).

При объявлении курсора можно указывать переменные PL/SQL, которые рассматриваются в качестве переменных привязки. Для курсоров справедливы обычные правила по определению области действия, поэтому эти переменные должны быть видимы в точке объявления курсора.

Для гарантии того, что все переменные, на которые производятся ссылки при объявлении курсора, уже объявлены, рекомендуется создавать все курсоры в конце раздела объявлений. Единственным исключением может быть ситуация, когда имя курсора указывается в ссылке — например, в атрибуте %ROWTYPE (при создании записи с полями, соответствующими списку выбора курсора). В этом случае курсор должен быть объявлен до того, как на него будет произведена ссылка.

Открытие курсора

Курсор открывается следующим образом:

OPEN имя_курсора;

где имя_курсора идентифицирует предварительно объявленный курсор.

Когда курсор открывается, происходит следующее:

• Анализируются значения переменных привязки.

• На основе значений переменных привязки и содержимого таблиц, к которым обращается запрос, определяется активный набор.

• Указатель активного набора устанавливается на первую строку.

В соответствии с транзакционной моделью запрос видит изменения, внесенные в базу данных до выполнения оператора OPEN, и изменения, сделанные в базе данных текущей транзакцией. Если другой сеанс изменил данные, но еще не зафиксировал эти изменения, они видны не будут.

Активный набор, т.е. набор строк, удовлетворяющих условию запроса, определяется во время открытия курсора (т.е. вычисляется табличное выражение).

Открытый курсор нельзя открыть еще раз. Если команда OPEN применяется к уже открытому курсору, Oracle генерирует ошибку ORA-6511.

Считывание строк из курсора

Предложение INTO запроса является частью оператора FETCH. Оператор FETCH имеет две формы:

FETCH имя_курсора INTO список_переменных;

FETCH имя_курсора INTO запись_PL/SQL;

где имя_курсора обозначает предварительно объявленный и открытый курсор, список_переменных представляет собой список предварительно объявленных переменных PL/SQL, разделенных запятыми, а запись_PL/SQL — это предварительно объявленная запись PL/SQL. В любом случае переменная (переменные) в конструкции INTO должна иметь тип, совместимый со списком выбора запроса.

В примере, приведенном ниже, демонстрируются правильный и неправильный операторы FETCH:

```
DECLARE
v_Department classes.department%TYPE;
v_Course classes.course%TYPE;
CURSOR c_AllClasses IS
SELECT *
FROM classes;
v_ClassesRecord c_AllClasses%ROWTYPE;
BEGIN
OPEN c_AllClasses;
- Это допустимый оператор FETCH, помещающий первую строку в запись
- PL/SQL, которая соответствует списку выбора запроса.
FETCH c_AllClasses INTO v_ClassesRecord;
- Этот оператор FETCH неверен, так как список выбора запроса
- возвращает 7 столбцов таблицы classes, но считывание
- происходит только в 2 переменные.
- Это приведет к ошибке "PLS-394: wrong number
- of values in the INTO list of a FETCH statement"
FETCH c_AllClasses INTO v_Department, v_Course;
END;
```

После каждого считывания FETCH указатель активного набора увеличивается и переходит к следующей строке. Таким образом, каждый оператор FETCH будет последовательно возвращать строки активного набора до тех пор, пока не будет передан весь набор.

Для определения момента считывания всего набора используется атрибут %NOTFOUND. При выполнении последнего считывания FETCH выходным переменным не будут присваиваться новые значения, т.е. они будут содержать прежние значения.

Закрытие курсора

После того как весь активный набор выбран, курсор следует закрыть. Это сообщает PL/SQL, что программа закончила работу с курсором и отведенные для него ресурсы могут быть освобождены. В состав этих ресурсов, входит пространство для хранения активного

набора, а также временное пространство, используемое для определения активного набора. Синтаксис закрытия курсора:

CLOSE имя_курсора;

где имя_курсора обозначает ранее открытый курсор.

После закрытия курсора считывать из него строки нельзя. Если попытаться сделать это, Oracle выдаст сообщение об ошибке:

Q ORA-1001: Invalid Cursor

(неверный курсор)

или

Q ORA-1002: Fetch out of Sequence

(непоследовательное считывание)

Попытка закрыть уже закрытый курсор также приведет к ошибке

ORA-1001.

После закрытия с курсором уже не будет связан активный набор. Однако в дальнейшем курсор вновь может быть открыт; при этом он снова получит активный набор — возможно, уже не такой, как раньше (в частности, если значения указанных в объявлении курсора переменных привязки к текущему моменту были изменены). Следует отметить, что изменение этих переменных при открытом курсоре не оказывает влияния на его активный набор.

Атрибуты курсора

В PL/SQL существуют четыре атрибута, которые могут быть применены к курсорам. Атрибуты курсора добавляются к имени курсора в блоке PL/SQL, подобно атрибутам %TYPE и %ROWTYPE. Однако курсорные атрибуты возвращают не тип, а значения, которые могут быть использованы в выражениях. К атрибутам курсора относятся %FOUND, %NOTFOUND, %ISOPEN и %ROWCOUNT.

%FOUND Это логический атрибут. Он возвращает TRUE, если при предшествующем считывании FETCH была извлечена строка, NULL – если курсор открыт, но не было произведено ни одного считывания строк и FALSE — в противном случае. Если курсор не открыт, то при проверке %FOUND выдается ошибка ORA-1001 (неверный курсор).

%NOTFOUND Ведет себя противоположно %FOUND: если предшествующее считывание возвращает строку — значение %NOTFOUND ложно. Атрибут %NOTFOUND возвращает TRUE, только если во время предшествующего считывания строка извлечена не была. Этот атрибут часто используется в качестве условия выхода из цикла выборки.

%ISOPEN Этот логический атрибут используется для определения, открыт или нет соответствующий курсор. Если курсор открыт, %ISOPEN возвращает TRUE, а если не открыт — FALSE.

%ROWCOUNT Этот числовой атрибут возвращает количество строк, считанных курсором на данный момент. Если %ROWCOUNT используется, когда соответствующий курсор еще не открыт, то возвращается ошибка ORA-1001.

Параметризованные курсоры

Существует еще один способ использования переменных привязки в курсоре. Параметризованные (parameterized) курсоры, подобно процедурам, принимают определенные аргументы. Рассмотрим курсор c_Classes:

```
DECLARE
v_Department classes.department%TYPE;
v_Course classes.course%TYPE;
CURSOR c_Classes IS
SELECT *
FROM classes
WHERE department = v_Department AND course = v_Course;
```

Курсор c_Classes содержит две переменные привязки — v_Department и v_Course. Можно превратить c_Classes в эквивалентный параметризованный курсор:

```
DECLARE
```

```
CURSOR c_Classes (p_Department classes.departnent%TYPE,
p_Course classes.course%TYPE) IS
SELECT *
FROM classes
WHERE department = p_Department
AND course = p_Course;
```

При работе с параметризованным курсором фактические значения указываются в операторе OPEN, например:

```
OPEN c_Classes('HIS', 101);
```

В этом случае 'HIS' передается как значение для p_Department, а 101 - для p_Course.

Обработка неявных курсоров

Явные курсоры используются для обработки тех операторов SELECT, которые возвращают более одной строки. Однако каждый оператор SQL выполняется в пределах контекстной области и поэтому имеет курсор, указывающий на конкретную контекстную область. Такой курсор называется SQL- курсором. В отличие от явных курсоров, SQL-курсор не открывается и не закрывается программой. PL/SQL неявно открывает SQL-курсор, обрабатывает SQL-оператор и затем закрывает этот курсор.

Неявные курсоры используются для обработки операторов INSERT, UPDATE, DELETE, а также однострочных операторов SELECT...INTO. SQL-курсор открывается и закрывается PL/SQL, поэтому команды OPEN, FETCH и CLOSE не нужны. Однако для SQL-курсоров можно применять курсорные атрибуты. Например, ниже приводится блок, в котором оператор INSERT выполняется в том случае, если оператору UPDATE не соответствует ни одной строки.

```
BEGIN
UPDATE rooms
SET number_seats = 100
WHERE room_id = 99980;
```

- Если оператор UPDATE не выбирает ни одной — строки, вводим новую строку в таблицу rooms.

```
IF SQL%NOTFOUND THEN
INSERT INTO rooms (room_id, number_seats)
VALUES (99980, 100);
END IF;
END;
```

Ту же задачу можно выполнить при помощи атрибута SQL%ROWCOUNT:

```
BEGIN
UPDATE rooms
SET number_seats = 100
WHERE room_id = 99980;
-- Если оператор UPDATE не выбирает ни одной
-- строки, вводим новую строку в таблицу rooms.
IF SQL%ROWCOUNT = 0 THEN
INSERT INTO rooms (room_id, number_seats)
VALUES (99980, 100);
END IF;
END;
```

Атрибут SQL%NOTFOUND можно использовать в операторах SELECT...INTO, но он мало подходит для этого. Дело в том, что если оператор SELECT...INTO не выбирает ни одной строки, Oracle выдает сообщение об ошибке ORA-1403: no data found

В результате управление блоком сразу же передается разделу обработки исключительных ситуаций, и атрибут SQL%NOTFOUND не проверяется.

Можно воспользоваться также атрибутом SQL%ISOPEN, однако он всегда будет возвращать FALSE, поскольку неявный курсор автоматически закрывается после обработки его оператора.

Циклы выборки курсора

Чаще всего курсоры используются для считывания всех строк активного набора с помощью цикла выборки. Цикл выборки (fetch loop) — это обычный цикл, в котором строки активного набора обрабатываются по порядку, одна за другой. Ниже рассматривается несколько различных типов циклов выборки курсоров и их применение.

Простые циклы

В этих циклах для обработки курсора используется синтаксис простого цикла (LOOP...END LOOP). Количество итераций задается с помощью атрибутов явного курсора. Рассмотрим пример:

```
DECLARE
- Объявим переменные для хранения информации о студентах,
- чьей профилирующей дисциплиной является история.
v_StudentID students.id%TYPE;
v_FirstName students. first_name%TYPE;
v_LastName students.last_name%TYPE;
- Курсор для считывания информации о студентах-историках
CURSOR c_HistoryStudents IS
SELECT id, first_name, last_name
FROM students
WHERE major = 'History';
BEGIN
-- Откроем курсор и инициализируем активный набор.
OPEN c_HistoryStudents;
LOOP
-- Считаем информацию о следующем студенте.
FETCH c_HistoryStudents INTO v_StudentID, v_FirstName, v_LastName;
-- Выйдем из цикла, если строк для выбора больше нет.
EXIT WHEN c_HistoryStudents%NOTFOUND;
-- Обработаем считанные строки. В нашем случае запишем всех
-- студентов в группу History 301, введя информацию о них в
-- таблицу registered_students. Кроме того, запишем имена и
-- фамилии в таблицу temp_table.
INSERT INTO registered_students (student_id, department, course)
VALUES (v_StudentID, 'HIS', 301);
INSERT INTO temp_table (num_col, char_col)
VALUES (v_StudentID, v_FirstName || ' ' || v_LastName);
END LOOP;
-- Освободим ресурсы, используемые курсором.
CLOSE c_HistoryStudents;
END;
```

Циклы WHILE

Циклы выборки курсоров можно построить также при помощи синтаксиса WHILE...LOOP. Рассмотрим пример:

```
DECLARE
-- Курсор для считывания информации о студентах-историках
CURSOR c_HistoryStudents IS
SELECT id, first_name, last_name
```

```
FROM students
WHERE major = 'History';
-- Объявим запись для хранения считанной информации.
v_StudentData c_HistoryStudents%ROWTYPE;
BEGIN
-- Откроем курсор и инициализируем активный набор.
OPEN c_HistoryStudents;
-- Считаем первую строку, чтобы установить цикл WHILE.
FETCH c_HistoryStudents INTO v_StudentData;
-- Выполняем цикл, пока есть строки для считывания.
WHILE cHistoryStudents%FOUND LOOP
```

-- Обработаем считанные строки. В нашем случае запишем всех студентов в группу History 301, введя информацию о них в таблицу registered_students. Кроме того, запишем имена и фамилии в таблицу temp_table.

```
INSERT INTO registered_students (student_id, department, course)
VALUES (v_StudentData. ID, 'HIS', 301);
INSERT INTO temp_table (num_col, char_col)
VALUES (v_StudentData.ID,
v_StudentData.first_name ‖ ' ' ‖ v_StudentData.last_name);
-- Считаем следующую строку. Условие %FOUND будет проверяться
-- перед тем, как цикл будет продолжен.
FETCH c_HistoryStudents INTO v_StudentData;
END LOOP;
-- Освободим ресурсы, используемые курсором.
CLOSE c_HistoryStudents;
END;
```

Этот цикл выборки действует точно так же, как и цикл LOOP...END

Курсорные циклы FOR

В обоих циклах выборки, описанных выше, необходимо обрабатывать курсоры явным образом с помощью операторов OPEN, FETCH и CLOSE.

В PL/SQL имеется упрощенный вид цикла — курсорный цикл FOR, в котором управление обработкой курсора осуществляется неявно. Рассмотрим пример:

```
DECLARE
-- Курсор для считывания информации о студентах-историках.
CURSOR c_HistoryStudents IS
SELECT id, first_name, last_name
FROM students
WHERE major = 'History';
BEGIN
-- Начнем цикл. Здесь производится неявное открытие курсора c_HistoryStudents.
FOR v_StudentData IN c_HistoryStudents LOOP
-- Здесь осуществляется неявное считывание.
-- Обработаем считанные строки. В нашем случае запишем всех
-- студентов в группу History 301, введя информацию о них в
-- таблицу registered_students. Кроме того, запишем имена и
-- фамилии в таблицу temp_table.
INSERT INTO registered_students (student_id, department, course)
VALUES (v_StudentData.ID, 'HIS', 301);
INSERT INTO temp_table (num_col, char_col)
VALUES (v_StudentData.ID,
v_StudentData.first_name | ' ' II v_StudentOata.last_name);
```

```
END LOOP;
-- После окончания цикла выполняется неявное закрытие
-- c_HistoryStudents.
END;
```

Необходимо отметить два важных аспекта, касающихся этого примера. Во-первых, запись v_StudentData не объявляется в разделе объявлений блока. Эта переменная неявно объявляется компилятором PL/SQL, подобно индексу цикла в числовых циклах FOR. Она имеет тип c_HistoryStudents%ROWTYPE, а ее область действия — цикл FOR. Неявное объявление индекса цикла и область действия этого объявления такие же, как и в числовом цикле FOR. Поэтому нельзя присваивать какое-либо значение переменной цикла внутри курсорного цикла FOR.

Во-вторых, курсор c_HistoryStudents открывается, считывается и закрывается неявным образом. Открывается он перед началом цикла. Перед каждой итерацией цикла атрибут %FOUND проверяется для установления наличия строк в активном наборе. Когда активный набор полностью выбран, курсор закрывается с окончанием цикла.

Курсорные циклы FOR хороши тем, что, обеспечивая функциональные возможности циклов выборки курсоров, они делают процесс считывания данных доступнее и понятнее, упрощая при этом синтаксис программы.

Неявные циклы FOR

Синтаксис цикла FOR можно сократить еще больше. Можно неявно объявить сам курсор, как показывает следующий пример:

```
BEGIN
-- Начало цикла. Здесь выполняется неявно OPEN.
FOR v_StudentData IN (SELECT id, first_name, last_name
FROM students
WHERE major = 'History') LOOP
-- Здесь выполняется неявно FETCH и проверяется %NOTFOUND.
-- Обработаем извлеченные строки. В нашем случае запишем всех студентов
-- в группу History 301, введя информацию о них в таблицу registered_students.
-- Кроме того, запишем имена и фамилии в таблицу temp_table.
INSERT INTO registered_students (student_id, department, course)
VALUES (v_StudentData.ID, 'HIS', 301);
INSERT INTO temp_table (num_col, char_col)
VALUES (v_StudentData.ID, v_StudentData.first_name || ' ' || v_StudentData.last_name);
END LOOP;
-- Теперь, когда цикл закончен, неявно выполняется CLOSE.
END;
```

Запрос содержится в скобках внутри самого оператора FOR. В данном случае запись v_StudentData и курсор объявляются неявно. Однако курсор не имеет имени.

Курсорные переменные

Во всех приведенных выше примерах явных курсоров рассматривались статические курсоры (static cursors), связанные с одним SQL-оператором, который был известен при компиляции блока. Курсорная же переменная (cursor variable) может быть связана с различными операторами во время выполнения программы. Курсорные переменные аналогичны переменным PL/SQL, в которых могут содержаться различные значения. Статические же курсоры аналогичны константам PL/SQL, так как они могут быть связаны только с одним запросом на этапе выполнения программы.

Чтобы воспользоваться курсорной переменной, ее необходимо предварительно объявить. Во время выполнения программы должна быть выделена память для хранения этой переменной, так как курсорные переменные имеют тип REF. Затем ее можно открывать,

считывать и закрывать так же, как статический курсор. Обычно курсорные переменные используются внутри хранимой процедуры, которая возвращает переменную клиентской программе. Эта техника позволяет хранимой процедуре открывать запрос и возвращать результирующий набор клиенту для обработки.

Объявление курсорной переменной

Курсорные переменные имеют ссылочный тип. Как говорилось ранее, ссылочный тип — это то же самое, что и указатель в языке программирования С. С помощью такого типа можно именовать области хранения данных во время выполнения программы. Чтобы воспользоваться ссылочным типом, необходимо сначала объявить переменную, а затем выделить область памяти. Ссылочные типы PL/SQL объявляются следующим образом:

REF тип

где тип — это предварительно определенный тип. Ключевое слово REF означает, что новый тип будет указателем на ранее определенный тип.

Таким образом, тип курсорной переменной — REF CURSOR. Полный синтаксис описания типа курсорной переменной таков:

TYPE имя_типа IS REF CURSOR [RETURN возвращаемый_тип];

где имя_типа — это имя нового ссылочного типа, а возвращаемый_тип — тип записи, указывающий типы списка выбора, которые в итоге будут возвращаться курсорной переменной.

Типом, возвращаемым курсорной переменной, должен быть тип записи. Запись может быть объявлена явно как определяемая пользователем или неявно при помощи %ROWTYPE. После определения ссылочного типа можно объявить переменную. Ниже приводится пример объявления различных курсорных переменных.

```
DECLARE
-- Описание при помощи %ROWTYPE
TYPE t_StudentsRef IS REF CURSOR
RETURN students%ROWTYPE;

-- Определяем новый тип записи,
TYPE t_NameRecord IS RECORD (
first_name students.first_name%TYPE,
last_name students. last_name%TYPE);
-- переменную этого типа
v_NameRecord t_NameRecord;
-- и курсорную переменную, использующую этот тип записи.
TYPE t_NamesRef IS REF CURSOR
RETURN t_NameRecord;

-- При помощи %TYPE можно объявить еще один тип.
TYPE t_NamesRef2 IS REF CURSOR
RETURN v_NameRecord%TYPE;

-- Объявим курсорные переменные, использующие созданные выше типы.
v_StudentCV t_StudentsRef;
v_NameCV t_NamesRef;
```

Ограниченные и неограниченные курсорные переменные

Курсорные переменные, рассмотренные в предыдущем разделе, являются ограниченными (constrained): они объявляются только для конкретного возвращаемого типа. Переменная должна открываться для такого запроса, список выбора которого соответствует типу, возвращаемому курсором. В противном случае возникает предопределенная исключительная ситуация ROWTYPE_MISMATCH (несоответствие типов строк).

Однако в PL/SQL разрешается объявлять неограниченные (unconstrained) курсорные переменные, для которых предложение RETURN отсутствует. Такая переменная может быть открыта для любого запроса. Ниже приводится раздел объявлений, в котором описывается неограниченная курсорная переменная.

```
DECLARE
- Определим неограниченный ссылочный тип
TYPE t_FlexibleRef IS REF CURSOR;
- и переменную этого типа.
v_CursorVar t_FlexibleRef;
```

Открытие курсорной переменной для запроса

Для связи курсорной переменной с определенным оператором SELECT используется расширенный синтаксис оператора OPEN, позволяющий указать требуемый запрос:

OPEN курсорная_переменная FOR оператор_select;

где курсорная_переменная — это ранее объявленная курсорная переменная,

а оператор_select — требуемый запрос. Если курсорная переменная ограничена, список выбора должен соответствовать типу, возвращаемому курсором. Иначе выдается сообщение об ошибке:

ORA-6504: PL/SQL: return types of result set variables or query do not match

В качестве примера рассмотрим курсорную переменную:

```
DECLARE
TYPE t_ClassesRef IS REF CURSOR RETURN classes%ROWTYPE;
v_ClassesCV t_ClassesRef;
BEGIN
--Теперь откроем v_ClassesCV:
OPEN v_ClassesCV FOR
SELECT * FROM classes;
END;
```

Если же попытаться открыть v_ClassesCV так:

```
OPEN v_ClassesCV FOR
SELECT department, course FROM classes;
```

то будет получено сообщение об ошибке ORA-6504, так как список выбора запроса не соответствует типу, возвращаемому этой курсорной переменной.

Оператор OPEN...FOR, в принципе, аналогичен оператору OPEN: анализируются переменные привязки, и определяется активный набор. После выполнения OPEN...FOR можно считывать информацию из курсорной переменной. Считывание может производиться с помощью оператора FETCH.

Закрытие курсорных переменных

Курсорные переменные закрываются точно так же, как статические курсоры, — при помощи оператора CLOSE. При этом освобождаются ресурсы, используемые запросом, однако память, отведенная для хранения самой курсорной переменной, освобождается не всегда, а только когда переменная выходит из области своего действия. Запрещается повторно закрывать ранее закрытые курсоры и курсорные переменные.

Ограничения на использование курсорных переменных

Курсорные переменные являются довольно мощным средством, и их применение намного упрощает обработку информации, поскольку они позволяют возвращать в одной переменной данные самых различных типов. Однако на их использование налагается ряд ограничений:

• Курсорные переменные нельзя объявлять в пакете. Сам тип можно, но переменную нельзя.

• Удаленные подпрограммы не могут возвращать значение курсорной переменной. Курсорные переменные могут передаваться между клиентской и серверной стороной PL/SQL (например, из клиента Oracle Form), но не между двумя серверами.

• Сборные конструкции PL/SQL (индексные таблицы, вложенные таблицы и изменяемые массивы) не могут хранить курсорные переменные. Аналогично, таблицы и представления базы данных не могут хранить столбцы REF CURSOR. Можно, однако, иметь массив клиентских курсорных переменных (такой, как JDBC ResultSets).

Процедуры, функции и пакеты

Как было указано ранее, существует два основных вида блоков PL/SQL: анонимные и именованные. Анонимные блоки (начинающиеся с DECLARE или с BEGIN) компилируются каждый раз при их использовании. Они не хранятся в базе данных, и их нельзя непосредственно вызывать из других блоков PL/SQL. Программные конструкции, которые рассматриваются в этой лекции, являются именованными блоками, они не имеют вышеназванных ограничений. Такие конструкции можно хранить в базе данных и исполнять по мере необходимости

Процедуры и функции

Процедуры и функции PL/SQL очень похожи на процедуры и функции, используемые в других языках третьего поколения, и обладают аналогичными свойствами. В совокупности процедуры и функции называются подпрограммами (subprogram).

После создания процедуры она сначала компилируется, а затем сохраняется в базе данных в скомпилированном виде. Скомпилированный код можно впоследствии выполнить из другого блока PL/SQL. Исходный код процедуры также сохраняется в базе данных.

Вызов процедуры — это оператор PL/SQL; в выражениях процедуры не вызываются. При вызове процедуры управление программой передается первому исполняемому оператору внутри нее. Когда процедура заканчивается, управление возвращается оператору, следующему за вызовом процедуры. В этом смысле процедуры PL/SQL функционируют точно так же, как и процедуры других языков третьего поколения. Функции вызываются в выражениях.

Создание подпрограмм

Как и другие объекты словаря данных, подпрограммы создаются оператором CREATE (процедуры — оператором CREATE PROCEDURE, а функции - оператором CREATE FUNCTION).

Синтаксис оператора создания процедуры:

CREATE [OR REPLACE] PROCEDURE имя_процедуры
[(аргумент [{IN | OUT |IN OUT}] тип
[, аргумент[{IN | OUT | IN OUT}] тип] …)] {IS | AS}
тело_ процедуры

где имя_процедуры — это имя создаваемой процедуры, аргумент — имя параметра процедуры, тип — это тип соответствующего параметра, а тело_процедуры — блок PL/SQL, в котором содержится код процедуры. Список аргументов является необязательным; в этом случае скобки отсутствуют как в объявлении, так и в вызове процедуры.

Чтобы изменить текст процедуры, необходимо удалить и повторно создать ее. Во время разработки процедур эта операция выполняется довольно часто, поэтому ключевые слова OR REPLACE (или заменить) позволяют выполнить такую операцию за один раз. Если процедура существует, она сначала удаляется безо всякого предупреждения. Если процедура до этого не существовала, то она создается. Если процедура существует, а ключевые слова OR REPLACE не указаны, то оператор CREATE возвращает ошибку Oracle: "ORA-955: Name is already used by an existing object".

Как и другие операторы CREATE, создание процедуры является операцией DDL, поэтому до и после создания процедуры неявно выполняются операторы COMMIT. При этом можно использовать как ключевое слово IS, так и ключевое слово AS — они эквивалентны друг другу.

Тело (body) процедуры — это блок PL/SQL, содержащий раздел объявлений, выполняемый раздел и раздел исключительных ситуаций. Раздел объявлений располагается между ключевым словом IS или AS и ключевым словом BEGIN; выполняемый раздел

(единственный обязательный) — между ключевыми словами BEGIN и EXCEPTION, а раздел исключительных ситуаций — между ключевыми словами EXCEPTION и END.

Следовательно, структура оператора создания процедуры такова:

```
CREATE OR REPLACE PROCEDURE имя_процедуры
[список_параметров] AS
/* Раздел объявлений */
BEGIN
/* Выполняемый раздел */
EXCEPTION
/* Раздел исключительных ситуаций */
END [имя_процедуры];
```

При объявлении процедуры ее имя можно при желании указать после последнего оператора END. Если после END идет идентификатор, он должен соответствовать имени процедуры.

Создание функции

Функции похожи на процедуры. И те, и другие получают некоторые параметры того или иного вида. Функции и процедуры — это различные формы блоков PL/SQL, в состав каждого из которых могут входить раздел объявлений, выполняемый раздел и раздел исключительных ситуаций. Как функции, так и процедуры можно хранить в базе данных или объявлять в блоке. Однако вызов процедуры является оператором PL/SQL, в то время как вызов функции — это часть некоторого выражения.

Синтаксис, применяемый при создании хранимой функции, похож на синтаксис создания процедуры:

```
CREATE [OR REPLACE] FUNCTION имя_функции
[(аргумент [{IN | OUT |IN OUT}] тип
[аргумент[{IN | OUT |IN OUT}] тип] …)]
RETURN возвращаемый_тип {IS | AS}
тело_функции
```

где имя_функции — это имя функции, аргумент и тип аналогичны аргументу и типу, указываемым при создании процедуры, возвращаемый_тип — это тип значения, возвращаемого функцией, а тело_функции — блок PL/SQL, содержащий программный код данной функции. Для тела функции применимы те же правила, что и для тела процедуры. Например, имя функции можно при желании указать после закрывающего END.

Как и для процедур, список аргументов необязателен. В этом случае ни при описании функции, ни при ее вызове круглые скобки указывать не нужно. Однако необходим тип, возвращаемый функцией, так как вызов функции является частью некоторого выражения.

Оператор RETURN. Внутри тела функции оператор RETURN используется для возврата управления программой в вызывающую среду с некоторым значением. Общий синтаксис оператора RETURN:

```
RETURN выражение;
```

где выражение — это возвращаемое значение. Значение выражения преобразуется в тип, указанный в команде RETURN при описании функции, если, конечно, это значение уже не имеет данный тип. При выполнении оператора RETURN управление программой сразу же возвращается в вызывающую среду.

В функции может быть несколько операторов RETURN, хотя выполняться будет только один из них. Отсутствие в функции оператора RETURN является ошибкой.

При использовании оператора RETURN в функции с ним должно быть связано некоторое выражение. Однако RETURN можно применять и в процедуре. В этом случае аргументы, приводящие к немедленной передаче управления в вызывающую среду, не указываются. Текущие значения формальных параметров, описанных как OUT или IN OUT, присваиваются фактическим параметрам, и выполнение программы продолжается с оператора, следующего за вызовом процедуры.

Ниже приведен пример вызова функции, возвращающей TRUE, если указанная учебная группа заполнена более чем на 80%, и FALSE в противном случае.

```
CREATE OR REPLACE FUNCTION AlmostFull (
p_Department classes.department%TYPE, p_Course classes.course%TYPE)
RETURN BOOLEAN IS
v_CurrentStudents NUMBER;
v_MaxStudents NUMBER;
v_ReturnValue BOOLEAN;
v_FullPercent CONSTANT NUMBER := 80;
BEGIN
- Получим текущее и максимальное число студентов в указанной группе.
SELECT current_students, max_students
INTO v_CurrentStudents, v_MaxStudents
FROM classes
WHERE department = p_Department AND course = p_Course;
- Если процент заполнения группы больше значения, заданного в
- v_FullPercent, возвращается TRUE. В противном случае
- возвращается FALSE.
IF (v_CurrentStudents / v_MaxStudents * 100) >= v_FullPercent THEN
v_ReturnValue := TRUE;
ELSE
v_ReturnValue := FALSE;
END IF;
RETURN v_ReturnValue;
END AlmostFull;
```

Функция AlmostFull возвращает логическое значение. Ниже приводится блок PL/SQL, в котором вызывается эта функция. Обратите внимание на то, что вызов функции не является оператором — он представляет собой фрагмент условного оператора IF, расположенного внутри цикла.

```
DECLARE
CURSOR c_Classes IS
SELECT department, course FROM classes;
BEGIN
FOR v_ClassRecord IN c_Classes LOOP
-- Выведем информацию обо всех группах, в которых осталось слишком мало места.
IF AlmostFull(v_ClassRecord.department,
v_ClassRecord.course) THEN
DBMS_OUTPUT.PUT_LINE(
v_ClassRecord.department ||'  '|| v_ClassRecord.course || ' is almost full!');
END IF;
END LOOP;
END;
```

Удаление процедур и функций

Как и таблицы, процедуры и функции могут быть удалены. При выполнении этой операции процедура или функция удаляется из словаря данных.

Синтаксис удаления процедуры выглядит следующим образом:

```
DROP PROCEDURE имя_процедуры;
```

Синтаксис удаления функции:

```
DROP FUNCTION имя_функции;
```

где имя_процедуры — имя существующей процедуры, а имя_функции — имя существующей функции.

Параметры подпрограмм

Параметры могут быть разного вида, и их разрешается передавать по значению или по ссылке.

Фактические параметры (actual parameter) содержат значения, передаваемые процедуре при ее вызове, и в них записываются результаты, возвращаемые процедурой (в зависимости от вида параметра). Именно значения фактических параметров используются в процедуре. Формальные параметры (formal parameter) указываются при объявлении процедуры или функции и выступают в роли места расположения фактических параметров. При вызове процедуры формальным параметрам присваиваются значения фактических параметров. Внутри процедуры все действия выполняются над формальными параметрами.

Формальные параметры бывают трех видов: IN, OUT и IN OUT. Если для формального параметра вид не указан, то по умолчанию устанавливается тип IN.

IN. Значение фактического параметра передается в процедуру при ее вызове. Внутри процедуры формальный параметр рассматривается в качестве константы PL/SQL, т.е. в качестве параметра только для чтения, и не может быть изменен. Когда процедура завершается и управление программой возвращается в вызывающую среду, фактический параметр не изменяется.

OUT. Любое значение, которое имеет фактический параметр при вызове процедуры, игнорируется. Внутри процедуры формальный параметр рассматривается в качестве неинициализированной переменной PL/SQL, т.е. содержит NULL- значение, и можно как записать в него значение, так и считать значение из него. Когда процедура завершается и управление программой возвращается в вызывающую среду, содержимое формального параметра присваивается фактическому параметру.

IN OUT. Этот вид представляет собой комбинацию IN и OUT. Значение фактического параметра передается в процедуру при ее вызове. Внутри процедуры формальный параметр рассматривается в качестве инициализированной переменной, и можно как записать в него значение, так и считать значение из него. Когда процедура завершается, и управление программой возвращается в вызывающую среду, содержимое формального параметра присваивается фактическому параметру.

Литералы и константы в качестве фактических параметров

Поскольку значения копируются, фактический параметр, соответствующий параметру IN OUT или OUT, должен быть переменной и не может быть константой или выражением. Должна существовать область для хранения возвращаемого значения.

Ограничения на формальные параметры

При вызове процедуры ей передаются значения фактических параметров, и внутри процедуры к этим значениям обращаются при помощи формальных параметров. При этом передаются не только значения, но и ограничения, наложенные на переменные, как часть механизма передачи параметров. При описании процедуры запрещается ограничивать длину параметров типа CHAR и VARCHAR2, а также точность и/или масштаб параметров типа NUMBER, поскольку ограничения принимаются от фактических параметров.

Исключительные ситуации, устанавливаемые в подпрограммах

В случае возникновения ошибки в подпрограмме устанавливается исключительная ситуация, которая может быть описана предварительно или определена пользователем. Если в процедуре отсутствует обработчик данной исключительной ситуации, управление программой сразу же передается из процедуры в вызывающую среду в соответствии с правилами, установленными для передачи исключительных ситуаций.

Однако в этом случае значения формальных параметров OUT и IN OUT не возвращаются фактическим параметрам. Фактические параметры будут иметь те же значения, которые они имели бы, если бы процедура не вызывалась.

Передача параметров по ссылке и по значению

Параметры подпрограмм передаются одним из двух способов: по ссылке или по значению. Когда параметр передается по ссылке (by reference), соответствующему формальному параметру передается указатель на фактический. Если же параметр передается по значению (by

value), оно копируется из фактического параметра в формальный. Передача по ссылке обычно осуществляется быстрее, так как при этом опускается операция копирования; особенно это касается параметров сборных конструкций (таблиц и изменяемых массивов). По умолчанию в PL/SQL параметры IN передаются по ссылке, а параметры IN OUT и OUT — по значению.

Использование NOCOPY

Существует специальное указание компилятору — NOCOPY. Синтаксис объявления параметра с этим модификатором таков:

имя_параметпра [вид] NOCOPY тип_данпых

где имя_параметра — это имя параметра, вид — вид параметра (IN, OUT или IN OUT), а тип данных — тип параметра. Если задано NOCOPY, компилятор PL/SQL пытается передать параметр по ссылке, а не по значению.

При указании NOCOPY для параметра IN генерируется ошибка компиляции, поскольку параметры IN всегда передаются по ссылке и NOCOPY к ним не применим.

Позиционное и именное представления

Во всех приведенных выше примерах фактические аргументы были связаны с формальными по позициям.

Рассмотрим пример.

```
CREATE OR REPLACE PROCEDURE CallMe (
p_ParameterA VARCHAR2,
p_ParameterB NUMBER,
p_ParameterC BOOLEAN,
p_ParameterD DATE) AS
BEGIN
NULL;
END CallMe;
DECLARE
v_Variable1 VARCHAR2(10);
v_Variable2 NUMBER(7,6);
v_Variable3 BOOLEAN;
v_Variable4 DATE;
BEGIN
CallMe(v_Variable1, v_Variable2, v_Variable3, v_Variable4);
END;
```

Можно видеть, что фактические параметры связаны с формальными по позициям: v_Variable1 с p_ParameterA, v_Variable2 с p_ParameterB и т.д. Это называется позиционным представлением (positional notation). Позиционное представление применяется наиболее часто, и именно оно используется в языках третьего поколения, например в C.

В качестве альтернативы можно вызвать процедуру при помощи именного представления (named notation):

```
DECLARE
v_Variable1 VARCHAR2(10);
v_Variable2 NUMBER(7,6);
v_Variables BOOLEAN;
v_Vanable4 DATE;
BEGIN
CallMe(p_ParameterA => v_Variable1,
p_ParameterB => v_Variable2,
p_ParameterC => v_Variable3,
p_ParameterD => v_Variable4);
END;
```

При именном представлении для каждого аргумента указываются как формальный, так и фактический параметры. Это дает возможность при желании установить собственный порядок аргументов. Например, в следующем блоке вызывается CallMe с теми же аргументами:

```
DECLARE
v_Variable1 VARCHAR2(10);
v_Variable2 NUMBER(7,6);
v_Variable3 BOOLEAN;
v_Variable4 DATE;
BEGIN
CallMe(p_ParameterB => v_Variable2,
p_ParameterC => v_Variable3,
p_ParameterD => v_Variable4,
p_ParameterA => v_Variable1);
END;
```

Кроме того, при необходимости позиционное и именное представления можно комбинировать в одном вызове. Первые аргументы нужно указывать по позициям, а оставшиеся можно указать по именам. Приведем пример:

```
DECLARE
v_Variable1 VARCHAR2(10);
v_Variable2 NUMBER(7,6);
v_Variables BOOLEAN;
v_Variable4 DATE;
BEGIN
- Первые 2 параметра передаются по позициям, а другие 2 - по именам.
CallMe(v_Variable1, v_Variable2,
p_ParameterC => v_Variable3,
p_ParameterD => v_Variable4);
END;
```

Именное представление — это еще одно свойство PL/SQL, которое заимствовано из языка Ada.

Значения параметров по умолчанию

Как и при объявлении переменных, формальные параметры процедуры или функции могут иметь значения по умолчанию. Если параметр имеет значение по умолчанию, то его можно не передавать из вызывающей среды. Если же параметр передается, то вместо значения по умолчанию берется фактический параметр. Значение по умолчанию для параметра указывается следующим образом:

имя_параметра [вид] тип_параметра

{:= | DEFAULT} исходное_значение

где имя_параметра — это имя формального параметра, вид — вид параметра (IN, OUT или IN OUT), тип_параметра — тип параметра (либо предварительно определенный, либо определяемый пользователем), а исходное_значение — значение, присваиваемое формальному параметру по умолчанию.

Можно применять символы := или ключевое слово DEFAULT.

Если применяется позиционное представление, то все параметры со значениями по умолчанию, не имеющие соответствующих фактических параметров, должны находиться в конце списка параметров. Рассмотрим следующий пример:

```
CREATE OR REPLACE PROCEDURE DefaultTest (
p_ParameterA NUMBER DEFAULT 10,
p_ParameterB VARCHAR2 DEFAULT 'abcdef',
p_ParameterC DATE DEFAULT SYSDATE) AS
BEGIN
DBMS_OUTPUT.PUT_LINE(
```

```
'A: ' || p_ParameterA ||
' B : ' || p_ParameterB ||
' C: ' || TO_CHAR(p_ParameterC, 'DD-MON-YYYY'));
END DefaultTest;
```

Все три параметра процедуры DefaultTest используют аргументы, заданные по умолчанию. Если нужно задать значение по умолчанию только для параметра p_ParameterB, а для параметров p_ParameterA и p_ParameterC требуется указать некоторые значения, необходимо использовать именное представление:

```
BEGIN
  DefaultTest(p_ParameterA => 7, p_ParameterC => '30-DEC-95');
END;
```

Если требуется использовать значение по умолчанию для параметра p_ParameterB и применять при этом позиционное представление, то для параметра p_ParameterC также нужно указать значение по умолчанию. При использовании позиционного представления все параметры по умолчанию, не имеющие соответствующих фактических параметров, должны находиться в конце списка параметров.

```
BEGIN
  DefaultTest(7);
END;
```

Если возможно, указывайте значения по умолчанию последними в списке аргументов. В этом случае можно использовать как позиционное, так и именное представление.

Оператор CALL

Оператор CALL может использоваться для вызова подпрограмм PL/SQL и Java с помощью оболочки PL/SQL и имеет синтаксис:

CALL имя_подпрограммы ([список_аргументов]) [INTO базовая_переменная];

где имя_подпрограммы — это автономная или модульная подпрограмма либо метод объектного типа, который может находиться в удаленной базе данных. Список_аргументов — разделенный запятыми список аргументов, а базовая_переменная используется для извлечения возвращаемого функцией значения.

• CALL является оператором SQL. Его нельзя использовать внутри блока PL/SQL, но можно использовать в динамическом SQL.

• Круглые скобки всегда должны присутствовать, даже если подпрограмма не имеет аргументов (или имеет используемые по умолчанию значения для всех аргументов).

• Предложение INTO применяется только для выходных переменных функций. Параметр IN OUT или OUT определяется как часть списка аргументов.

Пакеты

Пакет (package) — еще одно средство, пришедшее в PL/SQL из языка программирования Ada. Пакет — это конструкция PL/SQL, которая позволяет хранить связанные объекты в одном месте. Пакет состоит из двух частей: описания и тела. Они хранятся по отдельности в словаре данных.

Пакеты позволяют группировать связанные объекты.

В сущности, пакет представляет собой именованный раздел объявлений. Все, что может входить в состав раздела объявлений блока, может входить и в пакет: процедуры, функции, курсоры, типы и переменные. Размещение их в пакете полезно тем, что это позволяет обращаться к ним из других блоков PL/SQL, поэтому в пакетах можно описывать глобальные переменные PL/SQL (внутри одного сеанса работы с базой данных).

Описание пакета

В описании, или спецификации, пакета (package specification), называемом также заголовком пакета (package header), содержится информация о составе пакета, однако в описание не входит текст процедур. Рассмотрим пример:

```
CREATE OR REPLACE PACKAGE ClassPackage AS
```

-- Добавляет нового студента в указанную группу.
PROCEDURE AddStudent(p_StudentID IN students.id%TYPE,
p_Department IN classes.department%TYPE,
p_Course IN classes.course%TYPE);
- Удаляет указанного студента из указанной группы.
PROCEDURE RemoveStudent(p_StudentID IN students.id%TYPE,
p_Department IN classes.department%TYPE,
p_Course IN classes.course%TYPE);
-- Исключительная ситуация, устанавливаемая процедурой RemoveStudent.
e_StudentNotRegistered EXCEPTION;
-- Табличный тип, используемый для хранения информации о студентах.
TYPE t_StudentIDTable IS TABLE OF students. id%TYPE
INDEX BY BINARY_INTEGER;
- Возвращает таблицу PL/SQL со сведениями о студентах,
- включенных в указанную группу в настоящий момент.
PROCEDURE ClassList(p_Department IN classes.department%TYPE,
p_Course IN classes.course%TYPE,
p_IDs OUT t_StudentIDTable,
p_NumStudents IN OUT BINARY_INTEGER);
END ClassPackage;

В пакете ClassPackage содержатся три процедуры, тип и исключительная ситуация. Общий синтаксис создания заголовка пакета выглядит следующим образом:

CREATE [OR REPLACE] PACKAGE имя_пакета {IS | AS}
[определение_типа |
описание_процедуры |
описание_функции |
объявление_переменной |
объявление_исключителъной_ситуации |
объявленш_курсора |
объявление_прагмы] …
END [имя_пакета];

Элементы пакета (описания процедур и функций, переменные и т.д.) аналогичны тому, что указывается в разделе объявлений анонимного блока. Для заголовка пакета действуют те же синтаксические правила, что и для раздела объявлений, за исключением объявлений процедур и функций. Перечислим эти правила:

• Элементы пакета могут указываться в любом порядке. Однако, как и в разделе объявлений, объект должен быть объявлен до того, как на него будут произведены ссылки.

• Совсем не обязательно, чтобы присутствовали элементы всех видов. К примеру, пакет может состоять только из описаний процедур и функций и не содержать объявлений исключительных ситуаций или типов.

• Объявления всех процедур и функций должны быть предварительными. В предварительном объявлении (forward declaration) описываются подпрограмма и ее аргументы (если есть), но не включается программный текст. В этом отличие пакета от раздела объявлений блока, где могут находиться как предварительные объявления, так и текст процедур и функций. Программный текст процедур и функций пакета содержится в теле этого пакета.

Тело пакета

Тело пакета (package body) — это объект словаря данных, хранящийся отдельно от заголовка пакета. Тело пакета нельзя скомпилировать, если ранее не был успешно скомпилирован заголовок. В теле содержится текст подпрограмм, предварительно объявленных в заголовке пакета. В нем могут находиться также дополнительные объявления, глобальные для тела пакета, но не видимые в его описании. Тело пакета ClassPackage показано в следующем примере:

```
CREATE OR REPLACE PACKAGE BODY ClassPackage AS
-- Добавляет нового студента в указанную группу.
PROCEDURE AddStudent(p_StudentID IN students.id%TYPE,
p_Department IN classes.department%TYPE,
p_Course IN classes.course%TYPE) IS
BEGIN
INSERT INTO registered_students (student_id, department., course)
VALUES (p_StudentID, p_Department, p_Course);
END AddStudent;
- Удаляет указанного студента из указанной группы.
PROCEDURE RemoveStudent( p_StudetitID IN students. id%TYPE,
p_Department IN classes.department%TYPE,
p_Course IN classes.course%TYPE) IS
BEGIN
DELETE FROM registered_students
WHERE student_id = p_StudentID
AND department = p_Department
AND course = p_Course;
- Проверим, успешно ли была выполнена операция DELETE. Если
- указанные строки не найдены, устанавливается ошибка.
IF SQL%NOTFOUND THEN
RAISE e_StudentNotRegistered;
END IF;
END RemoveStudent;
- Возвращает таблицу PL/SQL со сведениями о студентах,
- включенных в указанную группу в настоящий момент.
PROCEDURE ClassList(pDepartment IN classes.department%TYPE,
p_Course IN classes.course%TYPE,
p_IDs OUT t_StudentIDTable,
p_NumStudents IN OUT BINARY_INTEGER) IS
v_StudentID registered_students.student_id%TYPE;
- Локальный курсор для выбора зарегистрированных студентов.
CURSOR c_RegisteredStudents IS
SELECT student_id
FROM registered_students
WHERE department = p_Department
AND course = p_Course;
BEGIN
/* p_NumStudents - индекс таблицы. Он будет
начинаться с 0 и увеличиваться с каждым циклом выборки.
В конце цикла в индексе будет содержаться число считанных
строк, т.е. число строк, возвращаемых в p_IDs. */
p_NumStudents := 0;
OPEN c_RegisteredStudents;
LOOP
FETCH c_RegisteredStudents INTO v_StudentID;
EXIT WHEN c_RegisteredStudents%NOTFOUND;
p_NumStudents := p_NumStudents + 1;
p_IDs(p_NumStudents) := v_StudentID;
END LOOP;
END ClassList;
END ClassPackage;
```

В теле пакета содержится программный текст для предварительных объявлений, сделанных в заголовке пакета, и могут также находиться дополнительные переменные, курсоры, типы и подпрограммы.

Тело пакета не является обязательной его частью. Если в заголовке не указаны какие-либо процедуры или функции (а только переменные, курсоры, типы и т.д.), тело можно не создавать. Такой способ полезен для объявления глобальных переменных, поскольку все объекты пакета видимы вне его пределов.

Любое предварительное объявление в заголовке пакета должно быть раскрыто в его теле. Описание процедуры или функции должно быть таким же и включать в свой состав имя подпрограммы, имена ее параметров и вид каждого параметра.

Пакеты и область действия

Любой объект, объявленный в заголовке пакета, находится в области действия и видим вне этого пакета. При обращении к объекту нужно указать имя пакета. Например, можно вызвать процедуру ClassPackage.RemoveStudent из блока PL/SQL:

```
BEGIN
  ClassPackage.RemoveStudent(10006, 'HIS', 101);
END;
```

Вызов этой процедуры аналогичен вызову процедуры, не входящей в состав пакета. Единственное отличие — указание перед именем процедуры имени пакета. Для пакетных процедур могут задаваться параметры по умолчанию, и вызывать такие процедуры можно при помощи как позиционного, так и именного представления, т.е. точно так же, как обычные хранимые процедуры.

Кроме того, в пакете можно применять типы данных, определяемые пользователем.

Внутри тела пакета на объекты, представленные в его заголовке, можно ссылаться без указания имени пакета. Тем не менее, при желании можно применять и полностью квалифицированные имена.

Область действия объектов, содержащихся в теле пакета

Процедуры, объявленные в теле пакета локально, имеют область действия, соответствующую только телу пакета. Их можно вызывать из других процедур тела пакета, но вне тела они невидимы.

Перегрузка пакетных подпрограмм

Внутри пакета процедуры и функции могут быть перегружены (overloaded).

Это означает, что может существовать несколько процедур или функций с одним и тем же именем, но с разными параметрами. Заметим, что обычные процедуры и функции (вне пакета) перегружать нельзя. Это очень удобно, так как позволяет выполнять одну и ту же операцию над объектами различных типов. Предположим, что нужно внести некоторого студента в состав одной из групп, указав либо идентификатор этого студента, либо его имя и фамилию. Это можно сделать, изменив ClassPackage следующим образом:

```
CREATE OR REPLACE PACKAGE ClassPackage AS
-- Добавляет нового студента в указанную группу.
PROCEDURE AddStudent(p_StudentID IN students.id%TYPE,
p_Department IN classes.department%TYPE,
p_Course IN classes.course%TYPE);
-- Также добавляет нового студента, но не по
-- идентификатору этого студента, а по его имени и фамилии.
PROCEDURE AddStudent(p_FirstName IN students.first_name%TYPE,
p_LastName IN students. last_name%TYPE,
p_Department IN classes.department%TYPE,
p_Course IN classes.course%TYPE);
END ClassPackage;
CREATE OR REPLACE PACKAGE BODY ClassPackage AS
-- Добавляет нового студента в указанную группу.
```

```
PROCEDURE AddStudent(p_StudentID IN students.id%TYPE,
p_Department IN classes.department%TYPE,
p_Course IN classes.course%TYPE) IS
BEGIN
  INSERT INTO registered_students (student_id, department, course)
  VALUES (p_StudentID, p_Department, p_Course);
END AddStudent;
-- Добавляет нового студента не по идентификатору, а по имени.
PROCEDURE AddStudent(p_FirstName IN students.first_name%TYPE,
p_LastName IN students.last_name%TYPE,
p_Department IN classes.department%TYPE,
p_Course IN classes.course%TYPE) IS
v_StudentID students.id%TYPE;
BEGIN
  /* Сначала получим нужный идентификатор в таблице students. */
  SELECT ID
  INTO v StudentID
  FROM students
  WHERE first_name = p_FirstName
  AND last_name = p_LastName;
  - Теперь добавим студента по его идентификатору.
  INSERT INTO registered_students (student_id, department, course)
  VALUES (v_StudentID, p_Department, p_Course);
END AddStudent;
END ClassPackage;
```

Теперь можно внести студента в группу Music 410 так:

```
BEGIN
  ClassPackage.AddStudent(10000, 'MUS', 410);
END;
```

или так:

```
BEGIN
  ClassPackage.AddStudent('Rita', 'Razmataz', 'MUS', 410);
END;
```

Перегрузка полезна тогда, когда одна и та же операция может быть выполнена над аргументами разных типов. Однако на перегрузку налагается ряд ограничений:

• Нельзя перегружать две подпрограммы, если их параметры отличаются только именами или видами. К примеру, следующие две процедуры не являются перегружаемыми:

```
PROCEDURE OverloadMe(p_TheParameter IN NUMBER);
PROCEDURE OverloadMe(p_TheParameter OUT NUMBER);
```

• Нельзя перегружать две функции, отличающиеся лишь типами возвращаемых ими данных. К примеру, следующие две функции не являются перегружаемыми:

```
FUNCTION OverloadMeToo RETURN DATE;
FUNCTION OverloadMeToo RETURN NUMBER;
```

• Наконец, типы параметров перегружаемых функций должны принадлежать различным семействам типов. Например, типы CHAR и VARCHAR2 входят в одно и то же семейство, поэтому не являются перегружаемыми следующие процедуры:

```
PROCEDURE OverloadChar(p_TheParameter IN CHAR);
PROCEDURE OverloadChar(p_TheParameter IN VARCHAR2);
```

Инициализация пакета

При первом вызове подпрограммы пакета или любом обращении к переменной или типу пакета создается его экземпляр (instantiated). Это значит, что пакет считывается с диска в

память, а затем запускается скомпилированный код вызванной подпрограммы. В этот момент для всех переменных, описанных в пакете, выделяется память. У каждого сеанса будет собственная копия пакетных переменных; это гарантирует, что два сеанса, выполняющие подпрограммы одного и того же пакета, будут использовать различные области памяти.

Во многих случаях при создании экземпляра пакета требуется запускать код инициализации. Для этого к телу пакета нужно добавить раздел инициализации, разместив его после всех объектов:

```
CREATE OR REPLACE PACKAGE BODY имя_модуля {IS | AS}
BEGIN
  код_инициализации;
END [имя_модуля];
```

где имя__модуля — это имя модуля, а код_инициализации — запускаемый код.

Исключительные ситуации

В основе PL/SQL лежит язык программирования Ada, одним из свойств которого является механизм исключительных ситуаций. При использовании этого механизма написанные на PL/SQL программы становятся гораздо надежнее, и во время их выполнения предоставляется возможность обработки как запланированных, так и незапланированных ошибок. Исключительные ситуации могут быть связаны с ошибками Oracle или с ошибками, определяемыми пользователем. Исключительные ситуации PL/SQL не являются объектами и не имеют методов.

Исключительные ситуации создаются только для обработки ошибок времени выполнения, но не для ошибок компиляции. Ошибки компиляции распознаются системой поддержки PL/SQL, и сообщения о них передаются пользователю. Такие ошибки программа обработать не может, поскольку на этом этапе она еще не выполняется.

Исключительные ситуации и их обработчики — это метод, при помощи которого программа реагирует на ошибки времени выполнения и устраняет их.

Объявление исключительных ситуаций

Исключительные ситуации описываются в разделе объявлений блока, инициируются в выполняемом разделе, а обрабатываются в разделе исключительных ситуаций. Существуют два вида исключительных ситуаций: определяемые пользователем и стандартные (предопределенные).

Исключительные ситуации, определяемые пользователем

Исключительная ситуация, определяемая пользователем, обозначает такую ошибку, которая описывается программистом, причем совсем не обязательно, чтобы эта ошибка была ошибкой Oracle, — она может быть, например, ошибкой данных. Стандартные же исключительные ситуации соответствуют типичным ошибкам SQL и PL/SQL.

Исключительные ситуации, определяемые пользователем, описываются в разделе объявлений блока PL/SQL. Как и переменные, исключительные ситуации имеют собственный тип (EXCEPTION) и область действия. Например:

```
DECLARE
  e_TooManyStudents EXCEPTION;
```

e_TooManyStudents — это идентификатор, который виден во всем блоке.

Область действия исключительной ситуации определяется так же, как и область действия переменной или курсора, описанного в том же разделе объявлений.

Стандартные исключительные ситуации

В Oracle существует ряд исключительных ситуаций, которые соответствуют типичным ошибкам Oracle. Как и стандартные типы данных (NUMBER, VARCHAR2 и т.д.), идентификаторы таких исключительных ситуаций описаны в модуле STANDARD. Эти идентификаторы доступны программе, и их не надо описывать в разделе объявлений, в отличие от исключительных ситуаций, определяемых пользователем.

Инициирование исключительных ситуаций

Когда возникает ошибка, связанная с некоторой исключительной ситуацией, инициируется (устанавливается) эта исключительная ситуация. Исключительные ситуации, определяемые пользователем, устанавливаются явно при помощи оператора RAISE, в то время как стандартные исключительные ситуации (или определенные пользователем исключительные ситуации, связанные с ошибкой Oracle посредством прагмы EXCEPTION_INIT) инициируются неявно при возникновении соответствующих ошибок Oracle. Если возникает ошибка Oracle, не связанная с исключительной ситуацией, тоже инициируется исключительная ситуация. Ее можно перехватывать с помощью обработчика OTHERS. Стандартные исключительные ситуации при желании можно устанавливать также с помощью оператора RAISE.

Пример.

```
DECLARE
  -- Исключительная ситуация для указания условия ошибки
  e_TooManyStudents EXCEPTION;
  -- Текущее число студентов, зарегистрированных в HIS-101
  v_CurrentStudents NUMBER(3);
  -- Максимальное число студентов, допустимое в HIS-101
  v_MaxStudents NUMBER(3);
BEGIN
  /* Определим текущее число зарегистрированных студентов и
  максимальное число студентов. */
  SELECT current_students, max_students
  INTO v_CurrentStudents, v_MaxStudents
  FROM classes
  WHERE department = 'HIS' AND course = 101;
  /* Сравним полученные значения. */
  IF v_CurrentStudents > v_MaxStudents THEN
    /* Зарегистрировано слишком много студентов - установим
    исключительную ситуацию.*/
    RAISE e_TooManyStudents;
  ENF IF;
END;
```

При установлении исключительной ситуации управление программой сразу же передается разделу исключительных ситуаций блока. Если такого раздела нет, исключительная ситуация передается блоку (распространяется на блок), в который входит данный блок. После передачи управления обработчику невозможно вернуться в выполняемый раздел блока.

Обработка исключительных ситуаций

При установлении исключительной ситуации управление программой передается разделу исключительных ситуаций блока. В этот раздел входят обработчики для некоторых или всех исключительных ситуаций. В обработчике содержится программный текст, выполняющийся при возникновении соответствующей ошибки и при установлении данной исключительной ситуации. Ниже приводится синтаксис этого раздела:

```
EXCEPTION
  WHEN имя_исключителъной_ситуауии1 THEN
    последовательностъ_операторов1;
  WHEN имя_исключителъной_ситуации2 THEN
    последовательности операторов2;
  [WHEN OTHERS THEN
    последователъностъ_операторов3;]
END;
```

Каждый обработчик состоит из условия WHEN и операторов, выполняющихся при установлении исключительной ситуации. В WHEN указывается, для какой исключительной ситуации предназначен данный обработчик. Продолжим рассмотренный выше пример:

```
DECLARE
 -- Исключительная ситуация для указания условия ошибки
 e_TooManyStudents EXCEPTION;
 -- Текущее число студентов, зарегистрированных в HIS-101
 v_CurrentStudents NUMBER(3);
 -- Максимально допустимое число студентов в HIS-101
 v_MaxStudents NUMBER(3);
BEGIN
 /* Определим текущее число зарегистрированных студентов и
 максимальное число студентов.*/
 SELECT current_students, max_students
 INTO v_CurrentStudents, v_MaxStudents
 FROM classes
 WHERE department = 'HIS' AND course = 101;
 /* Сравним полученные значения. */
 IF v_CurrentStudents > v_MaxStudents THEN
  /* Зарегистрировано слишком много студентов - установим
  исключительную ситуацию. */
  RAISE e_TooManyStudents;
 END IF;
EXCEPTION
 WHEN e_TooManyStudents THEN
  /*Обработчик, выполняющийся в том случае, если в HIS-101
  зарегистрировано слишком много студентов. Введем сообщение,
  поясняющее сложившуюся ситуацию.*/
  INSERT INTO log_table (info)
  VALUES ('History 101 has' ||v_CurrentStudents ||
  'students: max allowed is' || v_MaxStudents);
END;
```

Один обработчик может обслуживать несколько исключительных ситуаций, для чего нужно перечислить их имена в условии WHEN, отделив одно от другого ключевым словом OR (или):

```
EXCEPTION
 WHEN NO_DATA_FOUND OR TOO_MANY_ROWS THEN
  INSERT INTO log_table (info)
  VALUES ('A select error occured.');
END;
```

Данная исключительная ситуация может обрабатываться максимум одним обработчиком в разделе обработки исключений. Если для исключительной ситуации имеется более одного обработчика, компилятор PL/SQL будет инициировать ошибку PLS-483.

Обработчик исключений OTHERS

PL/SQL определяет специальный обработчик исключительных ситуаций — WHEN OTHERS. Обработчик OTHERS (другие) выполняется для всех инициированных исключительных ситуаций, которые не обработаны другими предложениями WHEN, определенными в текущем разделе исключений. Он всегда должен быть последним обработчиком в блоке, чтобы все предыдущие (и более специальные) обработчики были перед этим просмотрены. WHEN OTHERS будет перехватывать все исключения, предварительно определенные и определяемые пользователем. Обработчик OTHERS рекомендуется указывать на самом высоком уровне программы (в самом внешнем блоке) для обеспечения распознавания

всех возможных ошибок. Иначе ошибка будет распространяться в вызывающую среду. Это может привести к нежелательным последствиям, таким как откат текущей транзакции.

Добавим в рассматриваемый пример обработчик OTHERS:

```
DECLARE
  -- Исключительная ситуация для указания условия ошибки
  e_TooManyStudents EXCEPTION;
  -- Текущее число студентов, зарегистрированных в HIS-101
  v_CurrentStudents NUMBER(3);
  -- Максимально допустимое число студентов в HIS-101
  v_MaxStudents NUMBER(3);
BEGIN
  /*Определим текущее число зарегистрированных студентов и
  максимальное число студентов*/
  SELECT current_students, max_students
  INTO v_CurrentStudents, v_MaxStudents
  FROM classes
  WHERE department = 'HIS' AND course = 101;
  /*Сравним полученные значения.*/
  IF v_CurrentStudents > v_MaxStudents THEN
    /*Зарегистрировано слишком много студентов - установим
    исключительную ситуацию.*/
    RAISE e_TooManyStudents;
  END IF;
EXCEPTION
  WHEN e_TooManyStudents THEN
    /*Обработчик, выполняющийся в том случае, если в HIS-101
    зарегистрировано слишком много студентов. Введем сообщение,
    поясняющее сложившуюся ситуацию.*/
    INSERT INTO log_table (info)
    VALUES ('History 101 has' || v_CurrentStudents ||
    'students: max allowed is' || v_MaxStudents);
  WHEN OTHERS THEN
    /*Обработчик, выполняющийся для всех других ошибок.*/
    INSERT INTO log_table (info) VALUES ('Another error occurred');
  END;
```

При использовании обработчика OTHERS бывает полезно знать, какая ошибка Oracle установила исключительную ситуацию. Можно регистрировать не только факт возникновения ошибки, но и ее тип, если это необходимо для выполнения конкретных действий. В PL/SQL такие сведения получают при помощи двух встроенных функций: SQLCODE и SQLERRM. SQLCODE возвращает код текущей ошибки, а SQLERRM — текст сообщения об ошибке. Для исключений, определяемых пользователем, SQLCODE возвращает 1, а SQLERRM возвращает "User-defined Exception".

Ниже приводится полный текст блока PL/SQL, в котором используется обработчик исключительных ситуаций OTHERS.

```
DECLARE
  -- Исключительная ситуация для указания условия ошибки
  e_TooManyStudents EXCEPTION;
  -- Текущее число студентов, зарегистрированных в HIS-101
  v_CurrentStudents NUMBER(3);
  -- Максимально допустимое число студентов в HIS-101
  v_MaxStudents NUMBER(3);
  -- Переменная для хранения кода ошибки
```

```
  v_ErrorCode NUMBER;
  -- Переменная для хранения текста сообщения об ошибке
  v_ErrorText VARCHAR2(200);
BEGIN
  /*Определим текущее число зарегистрированных студентов и
  максимально допустимое число студентов.*/
  SELECT current_students, max_students
  INTO v_CurrentStudents, v_MaxStudents
  FROM classes
  WHERE department = 'HIS' AND course = 101;
  /*Сравним полученные значения.*/
  IF v_CurrentStudents > v MaxStudents THEN
    /*Зарегистрировано слишком много студентов - установим
    исключительную ситуацию.*/
    RAISE e_TooManyStudents;
  END IF;
EXCEPTION
  WHEN e_TooManyStudents THEN
    /*Обработчик, выполняющийся в случае, если в HIS-101
    зарегистрировано слишком много студентов. Введем сообщение,
    поясняющее сложившуюся ситуацию.*/
    INSERT INTO log_table (info)
    VALUES ('History 101 has' | v_CurrentStudents ||
    'students: max allowed is' || v_MaxStudents);
  WHEN OTHERS THEN
    /*Обработчик, выполняющийся для всех других ошибок*/
    v_ErrorCode := SQLCODE;
    -- Обратите внимание на использование SUBSTR
    v_ErrorText := SUBSTR(SQLERRM, 1, 200);
    INSERT INTO log_table (code, message, info) VALUES
    (v_ErrorCode, v_ErrorText, 'Oracle error occurred');
END;
```

Максимальная длина сообщения об ошибке Oracle составляет 512 символов. В примере переменная v_ErrorText ограничена 200 символами (для соответствия полю code таблицы log_table). (Второй параметр в функции Substr – начальная позиция, третий параметр – длина возвращаемой подстроки).

Функция SQLERRM может принимать числовой аргумент. В этом случае она возвращает текст сообщения об ошибке, код которой равен заданному числу. Аргумент должен всегда быть отрицательным.

При вызове в обработчике функция SQLCODE возвращает отрицательное число, обозначающее ошибку Oracle. Единственным исключением является ошибка "ORA-1403: no data found", когда SQLCODE возвращает +100.

Прагма EXCEPTION_INIT

Можно связывать именованные исключительные ситуации с конкретными ошибками Oracle, что позволяет обнаруживать эти ошибки непосредственно, а не с помощью обработчика OTHERS. Для этого служит прагма EXCEPTION_INIT. Прагма EXCEPTION_INIT используется следующим образом:

PRAGMA EXCEPTION_INIT (имя_исключителъной_ситуации,

номер_ ошибки_ Oracle);

где имя_исключмтелъной_ситуации — это имя исключительной ситуации, объявленной перед прагмой, а номер_ошибки_Orac1e — код ошибки, которую нужно связать с этой именованной исключительной ситуацией.

Прагма должна быть указана в разделе объявлений. Ниже приводится пример, в котором исключительная ситуация e_MissingNull определяется пользователем и устанавливается, если во время выполнения программы происходит ошибка "ORA-1400: mandatory NOT NULL column missing or NULL during insert" (при вводе данных в столбец NOT NULL пропущено значение или указано NULL-значение).

```
DECLARE
  e_MissingNull EXCEPTION;
  PRAGMA EXCEPTION_INIT(e_MissingNull, -1400);
BEGIN
  INSERT INTO students (id) VALUES (NULL);
EXCEPTION
  WHEN e_MissingNull THEN
    INSERT INTO log_table (info) VALUES ('ORA-1400 occurred');
END;
```

В одном предложении PRAGMA EXCEPTION_INIT можно связать с ошибкой Oracle только одну исключительную ситуацию, определяемую пользователем. В обработчике этой исключительной ситуации функции SQLCODE и SQLERRM будут возвращать код и сообщение, соответствующие ошибке Oracle, а не сообщение "User- Defined Exception" (определенное пользователем исключение).

Распространение исключительных ситуаций

Исключительные ситуации могут возникать в разделе объявлений, в выполняемом разделе или в разделе исключительных ситуаций блока PL/SQL. Выше было показано, что происходит в том случае, когда исключительная ситуация инициируется в выполняемом разделе блока и при этом имеется соответствующий обработчик. Но что произойдет, если обработчик отсутствует или исключительная ситуация возникает в другом разделе? Для ответа на этот вопрос следует рассмотреть процесс, называемый передачей (распространением) исключительных ситуаций (exception propagation).

Исключительные ситуации, возникающие в выполняемом разделе

Если исключительная ситуация возникает в выполняемом разделе блока PL/SQL, то для определения обработчика, который должен быть вызван, используется следующий алгоритм:

1. Если в текущем блоке имеется обработчик данной исключительной ситуации, он выполняется, блок успешно завершается и управление программой передается вышестоящему блоку.

2. Если обработчик отсутствует, исключительная ситуация передается в вышестоящий блок и инициируется там. После этого в вышестоящем блоке выполняется шаг 1. Если вышестоящего блока не существует, исключительная ситуация будет передана вызывающей среде, такой как SQL*Plus.

Исключительные ситуации, порождаемые в разделе объявлений

Если в операции присваивания раздела объявлений возникает исключительная ситуация, она немедленно передается охватывающему блоку. Здесь используются правила передачи исключительной ситуации, сформулированные в предыдущем разделе. Даже если в текущем блоке имеется обработчик этой исключительной ситуации, он не выполняется.

Исключительные ситуации, порождаемые в разделе исключительных ситуаций

Исключительные ситуации могут порождаться и в обработчиках исключительных ситуаций либо явно, посредством оператора RAISE, либо неявно при ошибке выполнения программы. В любом случае исключительная ситуация немедленно передается охватывающему блоку, также как и в случае раздела объявлений блока. Это происходит потому, что в разделе исключительных ситуаций в каждый конкретный момент времени активной может быть лишь одна исключительная ситуация. Пока она обрабатывается, может возникнуть другая, однако наличие одновременно нескольких исключительных ситуаций недопустимо.

Рекомендации по использованию исключительных ситуаций

Область действия исключительной ситуации

Область действия исключительной ситуации аналогична области действия переменной. Если исключительная ситуация, определенная пользователем, будет передана из блока и окажется вне области своего действия, ссылаться на нее по имени станет невозможно.

Отслеживание всех исключительных ситуаций

Не следует допускать возникновения в программах исключительных ситуаций, которые не обрабатываются. Можно воспользоваться обработчиком OTHERS, создав его на самом верхнем уровне программы.

Выявление места возникновения ошибки

Иногда трудно определить, какой из SQL-операторов стал причиной ошибки, так как раздел исключительных ситуаций анализируется для всего блока. Рассмотрим пример:

```
BEGIN
  SELECT . . .
  SELECT . . .
  SELECT . . .
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    -- Какой из операторов SELECT породил эту исключительную ситуацию?
END;
```

Решить эту проблему можно двумя способами. Первый — создание счетчика, указывающего на SQL-оператор:

```
DECLARE
  -- Переменная для хранения номера оператора выбора
  v_SelectCounter NUMBER := 1;
BEGIN
  SELECT .. .
  v_SelectCounter := 2;
  SELECT , . .
  v_SelectCounter := 3;
  SELECT .. .
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    INSERT INTO log_table (info) VALUES ('No data found in select' ||
    v_SelectCounter);
END;
```

Второй способ — размещение каждого оператора в собственном внутреннем блоке.

Сборные конструкции

PL/SQL позволяет оперировать одновременно несколькими переменными как единым целым. Мы уже рассмотрели один составной тип данных – записи.

Объявление и использование типов сборных конструкций

PL/SQL имеет два составных типа: записи и сборные конструкции. Запись позволяет интерпретировать несколько переменных, например все поля таблицы students, как целое. Записи аналогичны структурам в C. Сборные конструкции также являются составными типами данных в том смысле, что позволяют интерпретировать несколько переменных как целое. Однако вместо объединения нескольких переменных различных типов, сборная конструкция объединяет элементы одного и того же типа, аналогично массиву C или Java. Существуют три типа сборных конструкций: индексные таблицы (index-by tables), вложенные таблицы (nested tables) и изменяемые массивы (varray).

Вложенные таблицы можно хранить в таблицах базы данных (именно поэтому они называются вложенными) и использовать непосредственно в SQL. Индексные же таблицы

существуют исключительно в PL/SQL, и их нельзя хранить непосредственно в таблице базы данных. В совокупности индексные таблицы и вложенные таблицы называют таблицами PL/SQL. Изменяемые массивы сходны с таблицами PL/SQL способом обращения к ним. Однако изменяемые массивы объявляются с фиксированным числом элементов, в то время как для таблиц PL/SQL верхний предел не устанавливается.

Индексные таблицы

Чтобы объявить индексную таблицу, сначала нужно определить ее тип в блоке PL/SQL, а затем объявить переменную данного типа (так же, как и в случае записей). Общий синтаксис описания индексной таблицы таков:

TYPE тип_таблицы IS TABLE OF тип INDEX BY BINARY_INTEGER;

где тип_таблицы — имя нового типа, а тип — предопределенный тип либо ссылка на тип посредством %TYPE или %ROWTYPE.

Предложение INDEX BY BINARY_NTEGER обязательно в описании индексной таблицы, но не требуется для вложенных таблиц.

Приведем примеры объявления различных типов таблиц и переменных PL/SQL.

```
DECLARE
TYPE NameTab IS TABLE OF students.first_name%TYPE
INDEX BY BINARY_INTEGER;
TYPE DateTab IS TABLE OF DATE
INDEX BY BINARY_INTEGER;
v_Names NameTab;
v_Dates DateTab;
```

После того как объявлены тип и переменная, можно ссылаться на отдельные элементы таблицы PL/SQL следующим образом:

имя_таблицы(индекс)

где имя_таблицы — это имя таблицы, а индекс — либо переменная, имеющая тип BINARY_NTEGER, либо переменная или выражение, которое может быть преобразовано в тип BINARY_INTEGER. Продолжим наш блок PL/SQL:

```
BEGIN
  v_Names(1) ;= 'Scott';
  v_Dates(-4) := SYSDATE - 1;
END;
```

Индексная таблица схожа с таблицей базы данных и содержит два столбца: key (ключ) и value (значение). Тип ключа — BINARY_NTEGER, а тип значения — это тип данных, указанный в описании.

При работе с индексными таблицами обращайте внимание на следующее:

• Число строк индексной таблицы может быть любым. Единственное ограничение (кроме доступного объема памяти) — это число значений ключа, которые представляются типом BINARY_INTEGER (-2147483647...+2147483647).

• Порядок элементов индексной таблицы необязательно должен быть строго определен. Эти элементы хранятся в памяти не подряд, как в массивах, поэтому они могут вводиться произвольно.

• Единственный тип данных, разрешенный для ключей, — это тип BINARY_NTEGER.

Несуществующие элементы

Присвоение значения несуществующему i-му элементу индексной таблицы создает этот элемент, что напоминает операцию INSERT, выполняемую над таблицей базы данных. Обращение к i-му элементу похоже на операцию SELECT. Действительно, если ссылка на i-ый элемент производится до того, как он создан, система поддержки PL/SQL возвращает сообщение об ошибке "ORA-1403: no data found" (данные не найдены), как и в случае таблицы базы данных.

Элементы индексной таблицы удаляются с помощью метода DELETE.

Индексные таблицы записей

В следующем примере представлена индексная таблица, состоящая из записей:

```
DECLARE
TYPE StudentTab IS TABLE OF students%ROWTYPE
INDEX BY BINARY_INTEGER;
/* Каждый элемент v_Students является записью. */
v_Students StudentTab;
BEGIN
/* Извлекаем запись с идентификатором 10001 и сохраняем ее в v_Students(10001). */
SELECT *
INTO v_Students(10001)
FROM students
WHERE id = 10001;
/* Присвоим значения v_Students(1) напрямую. */
v_Students(1).first_name := 'Ivan';
v_Students(1).last_name := 'Smirnov';
END;
```

Каждый элемент этой таблицы является записью, поэтому можно обращаться к полям данной записи следующим образом:

таблица(индекс). Поле

Вложенные таблицы

Базовые функциональные возможности вложенных таблиц ничем не отличаются от возможностей индексных таблиц. Вложенную таблицу можно рассматривать как таблицу базы данных, содержащую два столбца: столбец ключей и столбец значений. Из вложенной таблицы, как и из индексной, можно удалить любые элементы, после чего та становится разреженной, с непоследовательными ключами. Однако при создании вложенных таблиц ключи должны располагаться по порядку, и ключи не могут быть отрицательными.

Синтаксис создания вложенной таблицы:

TYPE имя_таблицы IS TABLE OF тип_таблицы [NOT NULL];

где имя_таблицы — это имя нового типа, а тип_таблицы — тип элементов вложенной таблицы. В качестве типа таблицы может использоваться объектный тип, определяемый пользователем, или выражение, в котором применяется %TYPE, но не могут использоваться BOOLEAN, NCHAR, NCLOB, NVARCHAR2 и REF CURSOR. Если указано NOT NULL, то элементы вложенной таблицы не могут быть NULL.

Единственное синтаксическое отличие индексной таблицы от вложенной заключается в указании в операторе создания таблицы предложения INDEX BY BINARY_NTEGER. Если это предложение отсутствует, то создается вложенная таблица, а если присутствует — то индексная таблица.

```
DECLARE
- Создадим тип вложенной таблицы на основе объектного типа,
TYPE ObjectTab IS TABLE OF MyObject;
-- Тип вложенной таблицы, основанный на %ROWTYPE
TYPE StudentsTab IS TABLE OF students%ROWTYPE;
- Переменные, которые имеют типы, созданные выше
v_ClassList StudentsTab;
v_ObjectList ObjectTab;
```

Инициализация вложенных таблиц

При создании индексной таблицы без элементов она будет пустой. А при объявлении вложенной таблицы без элементов (см. предыдущий пример) она, подобно другим типам PL/SQL, инициализируется с помощью NULL. При попытке добавить элемент к вложенной NULL-таблице возвращается сообщение об ошибке "ORA-6531: Reference to uninitialized

collection" (ссылка на неинициализированную сборную конструкцию). Эта ошибка соответствует предопределенной исключительной ситуации COLLECTION_IS_NULL. При выполнении следующего раздела возникнет ошибка:

```
BEGIN
- При выполнении этой операции присваивания будет установлена
- исключительная ситуация COLLECTION_IS_NULL, так как v_ObjectList
- является NULL.
v_ObjectList(1) :=
MyObject(-17, 'Goodbye', TO_DATE('01-01-200V, 'DD-MM-YYYY'));
END;
```

Инициализация вложенной таблицы делается при помощи функции-конструктора. Конструктор вложенной таблицы имеет то же имя, что и сама таблица. Однако число его аргументов варьируется, причем каждый из аргументов должен иметь тип, совместимый с типом таблицы. Аргументы становятся элементами таблицы с последовательными индексами, начиная с индекса 1.

```
DECLARE
TYPE NumbersTab IS TABLE OF NUMBER;
-- Создадим таблицу с одним элементом.
v_Tab1 NumbersTab := NumbersTab(-1);
-- Создадим таблицу с пятью элементами.
v_Primes NumbersTab := NunbersTab(1, 2, 3, 5, 7);
-- Создадим таблицу без элементов.
v_Tab2 NumbersTab := NumbersTab();
BEGIN
-- Присвоим значение v_Tab1(1). При этом значение, заданное
-- для v_Tab(1) при инициализации (-1), будет заменено.
v_Tab1(1) := 12345;
-- Распечатаем содержимое v_Primes.
FOR v_Count IN 1..5 LOOP
DBMS_OUTPUT.PUT(v_Primes(v_Count) || ' ');
END LOOP;
DBMS_OUTPUT.NEW_LINE;
END;
```

1 2 3 5 7

Пустые таблицы Обратите внимание на объявление v_Tab2 в предыду-

щем блоке. При этом создается таблица без элементов, однако она не является атомарной NULL-таблицей.

Добавление элементов в существующую таблицу

Хотя для таблицы не устанавливается никаких ограничений, нельзя присвоить значение элементу, который еще не существует. Если попытаться это сделать, PL/SQL выдаст сообщение об ошибке "ORA-6533: Subscript beyond count" (неправильный индекс), которая эквивалентна предопределенной исключительной ситуации SUBSCRIPT_BEYOND_COUNT.

Приведем пример:

```
DECLARE
TYPE NumbersTab IS TABLE OF NUMBER;
v_Numbers NumbersTab := NumbersTab(1, 2, 3);
BEGIN
-- v_Numbers инициализирована как состоящая из 3 элементов.
-- Поэтому следующие операции присваивания правильны.
v_Numbers(1) := 7;
v_Numbers(2) := -1;
```

```
-- Однако эта операция присваивания вызывает ORA-6533.
v_Numbers(4) := 4;
END;
```

Можно увеличить размер вложенной таблицы при помощи метода EXTEND (см. ниже).

Изменяемые массивы

Изменяемый массив, или массив с переменной длиной (varray, varying array, variable length array), — это тип данных, практически идентичный массиву в языке программирования C или Java. Синтаксически обращение к изменяемому массиву происходит так же, как к вложенной или индексной таблице. Однако для размера массива устанавливается фиксированная верхняя граница, указываемая в объявлении типа.

Структура массива — это не разреженная структура данных; элементы вводятся в массив, начиная с индекса 1 и до максимального значения, заданного в объявлении типа изменяемого массива.

Максимальный размер изменяемого массива — 2 гигабайта.

Изменяемый массив хранится так же, как массив в C или Java, — непрерывно в памяти, в отличие от вложенной или индексной таблицы, которая больше похожа на таблицу базы данных.

Объявление изменяемого массива

Тип изменяемого массива объявляется следующим образом:

```
TYPE имя_типа IS {VARRAY | VARYING ARRAY} (максималъный_размер)
OF тип_элементов [NOT NULL];
```

где имя_типа — это имя нового типа изменяемого массива, максимальный_размер — целое число, определяющее максимальное количество элементов в изменяемом массиве, а тип_элементов — скалярный тип, тип записи или объектный тип PL/SQL. Кроме того, тип элементов можно указать при помощи %TYPE, но недопустимы BOOLEAN, NCHAR, NCLOB, NVARCHAR2, REF CURSOR.

```
DECLARE
-- Несколько правильных типов изменяемых массивов.
-- Это список чисел, каждое из которых не должно быть null.
TYPE NumberList IS VARRAY(10) OF NUMBER(3) NOT NULL;
-- Список записей PL/SQL.
TYPE StudentList IS VARRAY(100) OF students%ROWTYPE;
-- Список объектов.
TYPE ObjectList IS VARRAY(25) OF MyObject;
```

Инициализация изменяемых массивов

Как и таблицы, изменяемые массивы инициализируются с помощью конструкторов. Число аргументов, передаваемых конструктору, становится начальной длиной изменяемого массива и не должно превышать максимальной длиной, указанной в его типе.

```
DECLARE
-- Определим тип изменяемого массива.
TYPE Numbers IS VARRAY(20) OF NUMBER(3);
-- Объявим изменяемый массив значений NULL.
v_NullList Numbers;
-- В этом массиве содержатся 2 элемента.
v_List1 Numbers := Numbers(1, 2);
-- В этом массиве содержится 1 элемент, равный NULL
v_List2 Numbers := Numbers(NULL);
BEGIN
IF v_NullList IS NULL THEN
DBMS_OUTPUT.PUT_LINE('v_NullList is NULL');
END IF;
```

```
IF v_List2(1) IS NULL THEN
DBMS_OUTPUT.PUT_LINE('v_List2(1) is NULL');
END IF;
END;
v_NullList is NULL
v_List2(1) is NULL
```

Работа с элементами изменяемых массивов

Как и в случае вложенной таблицы, начальный размер изменяемого массива определяется числом элементов, указываемых в конструкторе при объявлении массива. Присвоение значений элементам, не попадающим в указанный диапазон, приводит к ошибке "ORA-6533: Subscript beyond count" (неправильный индекс).

```
DECLARE
TYPE Strings IS VARRAY(5) OF VARCHAR2(10);
-- Объявим изменяемый массив, состоящий из трех элементов.
v_List Strings :=Strings('0ne', 'Two', 'Three');
BEGIN
-- Значение индекса в диапазоне от 1 до 3, поэтому
-- данная операция присваивания верна.
v_List(2) := 'TWO';
-- Значение вне диапазона; устанавливается ORA-6533.
v_List(4) := '!!!';
END;
```

Подобно вложенным таблицам, размер изменяемого массива можно увеличить при помощи метода EXTEND (см. ниже).

Однако в отличие от вложенных таблиц изменяемый массив не может увеличиться сверх того размера, который был указан как максимальный при объявлении типа изменяемого массива.

При попытке присвоить значения элементам вне максимального размера изменяемого массива или расширить его сверх максимального размера устанавливается ошибка "ORA-6532: Subscript outside of limit" (неверный индекс), которая эквивалентна предопределенной исключительной ситуации SUBSCRIPT_OUTSIDE_LIMIT.

Многоуровневые сборные конструкции

Все приведенные выше примеры содержали одномерные сборные конструкции. Допускаются сборные конструкции с большим числом измерений, т.е. сборные конструкции сборных конструкций. Это называется "многоуровневые сборные конструкции". Объявление типа для многоуровневых сборных конструкций такое же, как и для одномерных сборных конструкций, за исключением того, что тип сборной конструкции сам является сборной конструкцией. Следующий раздел объявлений показывает некоторые типы многоуровневых сборных конструкций.

```
DECLARE
-- Объявим сначала индексную таблицу чисел
TYPE t_Numbers IS TABLE OF NUMBER
INDEX BY BINARY_INTEGER;
-- Теперь объявим тип, который будет индексной таблицей t_Numbers.
-- Это многоуровневая сборная конструкция.
TYPE t_MultiNumbers IS TABLE OF t_Numbers
INDEX BY BINARY_INTEGER;
-- Можно также получить изменяемый массив индексных таблиц
TYPE t_MultiVarray IS VARRAY(10) OF t_Numbers;
-- Или вложенную таблицу
```

```
TYPE t_MultiNested IS TABLE OF t_Numbers;
v_MultiNumbers t_MultiNumbers;
```

Элемент многоуровневой сборной конструкции сам является сборной конструкцией, поэтому используются два набора скобок для доступа к элементу внутренней сборной конструкции:

```
BEGIN
vMultiNumbers (1) (1) := 12345;
END;
```

Сравнение типов сборных конструкций

В этом разделе обсуждаются сходства и различия рассмотренных выше трех типов сборных конструкций. Свойства этих типов сведены в таблицу.

Индексные таблицы	Вложенные таблицы	Изменяемые массивы
Нельзя хранить в таблицах базы данных	Можно хранить в таблицах базы данных	Можно хранить в таблицах базы данных
Ключи могут быть положительными или отрицательными	Ключи должны быть положительными	Ключи должны быть положительными
Не имеют явно заданного максимального размера	Имеют явно заданный максимальный размер	Ограничены максимальным размером, указываемым в определении типа
Могут быть разреженными с непоследовательными значениями ключа	Могут быть разреженными с непоследовательными значениями ключа	Всегда выделяется область для каждого элемента; значения ключа последовательные
Не могут быть атомарным NULL	Могут быть атомарным NULL	Могут быть атомарным NULL
Можно объявлять только в блоках PL/SQL	Можно объявлять в блоках PL/SQL или вне их с помощью CREATE TYPE	Можно объявлять в блоках PL/SQL или вне их с помощью CREATE TYPE
Значения элементам присваиваются напрямую, без инициализации	Перед присвоением значений элементам таблица должна быть инициализирована	Перед присвоением значений элементам массив должен быть инициализирован

Сборные конструкции в базе данных

В рассмотренных выше примерах сборные конструкции обрабатывались в блоках PL/SQL. Однако вложенные таблицы и изменяемые массивы (но не индексные таблицы) можно хранить еще и в таблицах базы данных.

Хранимые сборные конструкции базы данных различаются по способу объявления табличных типов и по синтаксису создания таблиц со столбцами, имеющими тип сборной конструкции.

Для записи сборной конструкции в таблицу базы данных и для считывания ее оттуда необходимо, чтобы тип конструкции был известен и в PL/SQL, и в SQL. Для этого тип конструкции нужно объявить с помощью оператора CREATE TYPE Например:

```
CREATE OR REPLACE TYPE NameList AS
VARRAY(20) OF VARCHAR2(30);
DECLARE
-- Поскольку NameList является глобальным для PL/SQL, можно сослаться на
-- него и в другом блоке.
v_Names2 NameList;
BEGIN
```

NULL;
END;

Тип, создаваемый на уровне схемы (оператором CREATE OR REPLACE TYPE), считается глобальным для PL/SQL, и правила для областей его действия и видимости те же, что и для любого другого объекта базы данных. Кроме того, тип уровня схемы можно назначить столбцу базы данных.

Тип же, объявленный локальным в блоке PL/SQL, виден только в этом блоке и не доступен для столбцов базы данных. Тип, объявленный в заголовке модуля, виден во всем PL/SQL, но для столбцов тем не менее не доступен. Столбцам базы данных можно назначать только типы уровня схемы.

Структура хранимых изменяемых массивов

Изменяемый массив можно использовать в качестве типа для столбца базы данных. В этом случае весь массив хранится в одной строке базы среди других столбцов. В разных строках содержатся разные изменяемые массивы. Рассмотрим следующие объявления:

```
CREATE OR REPLACE TYPE BookList AS VARRAY(10) OF NUMBER(4);
CREATE TABLE class_material (
department CHAR(3),
course NUMBER(3),
required_reading BookList);
```

Данные изменяемого массива, которые занимают больше 4 Кбайт, в действительности хранятся отдельно от остальных столбцов таблицы в LOB. Можно определить параметры хранения LOB отдельно в операторе CREATE TABLE.

В таблице class_material содержатся номера книг, обязательных для чтения в данной группе. Этот список хранится как столбец типа изменяемого массива. Тип любого столбца тип изменяемого массива должен быть известен в базе данных и должен храниться в словаре данных, поэтому требуется оператор CREATE TYPE.

Структура хранимых вложенных таблиц

Как и изменяемые массивы, вложенные таблицы могут храниться в виде столбцов базы данных. В каждой строке таблицы базы данных может содержаться отдельная вложенная таблица. В качестве примера смоделируем каталог библиотеки. Сделаем это с помощью следующих определений:

```
CREATE OR REPLACE TYPE StudentList AS TABLE OF NUMBER(5);
CREATE TABLE library_catalog (
catalog_number NUMBER(4),
FOREIGN KEY (catalog_number) REFERENCES books(catalog_number),
num_copies NUMBER,
num_out NUMBER,
checked_out StudentList)
NESTED TABLE checked_out STORE AS co_tab;
```

В таблице library_catalog четыре столбца, в том числе столбец номеров книг, и вложенная таблица с идентификаторами студентов, получивших экземпляры книг. По поводу хранения вложенных таблиц следует сказать несколько слов:

• Табличный тип используется в определении таблицы точно так же, как и объектный или встроенный тип столбца. Это должен быть тип уровня схемы, создаваемый оператором CREATE TYPE.

Для каждой вложенной таблицы в конкретной таблице базы данных необходимо использовать предложение NESTED TABLE, которое определяет имя таблицы хранения.

Таблица хранения (store table) — это таблица, которая создается системой и используется для хранения фактических данных вложенной таблицы. В отличие от изменяемых массивов данные вложенной таблицы хранятся отдельно, а не среди остальных столбцов таблицы. Реально в столбце checked_out будет содержаться ссылка (REF) на таблицу co_tab, в которой и находится список идентификаторов студентов.

Таблица хранения может быть описана, и она существует в user_tables, но обратиться непосредственно к ней нельзя. При попытке ее модификации или обращения к ней с запросом возвращается ошибка Oracle "ORA-22812: Cannot reference nested table column's storage table". Работа с содержимым таблицы хранения осуществляется посредством SQL-операторов, выполняемых над основной таблицей.

Манипуляции со сборными конструкциями

С помощью SQL-операторов DML можно выполнять манипуляции над хранимой сборной конструкцией. Операции этого типа воздействуют на сборную конструкцию в целом, а не на отдельные элементы. Для работы с элементами сборной конструкции используется PL/SQL, а также операторы SQL

INSERT

Для ввода сборной конструкции в строку базы данных используется оператор INSERT. Сборная конструкция должна быть предварительно создана и инициализирована. Кроме того, она может быть переменной PL/SQL.

```
DECLARE
-- BookList – изменяемый массив
v_CSBooks Booklist := BookList(1000, 1001, 1002);
v_HistoryBooks Booklist := BookList(2001);
BEGIN
- В INSERT используется вновь создаваемый изменяемый массив из 2 элементов.
INSERT INTO classjnaterial
VALUES CMUS', 100, BookList(3001, 3002));
- В INSERT используется ранее инициализированный изменяемый массив из
-- 3 элементов.
INSERT INTO classjnaterial VALUES CCS', 102, v_CSBooks);
- В INSERT используется ранее инициализированный изменяемый массив из 1 элемента.
INSERT INTO classjnaterial VALUES ('HIS1, 101, v_HistoryBooks);
END;
```

UPDATE

Для модификации хранимых сборных конструкций используется оператор UPDATE. После выполнения следующего примера таблица library_catalog будет выглядеть, как показано на рис. 8.2:

```
DECLARE
v_StudentList1 Studentlist := Studentlist(10000, 10002, 10003);
v_Studentlist2 Studentlist := Studentlist( 10000, 10002, 10003);
v_Studentlist3 Studentlist := Studentlist(10000, 10002, 10003);
BEGIN
- Сначала введем строки с вложенными NULL-таблицами.
INSERT INTO library_catalog (catalog_number, num_copies, num_out)
VALUES (1000, 20, 3);
INSERT INTO library_catalog (catalog_number, num_copies, num_out)
VALUES (1001, 20, 3);
INSERT INTO library_catalog (catalog_number, num_copies, num_out)
VALUES (1002, 10, 3);
INSERT INTO library_catalog (catalog_number, num_copies, num_out)
VALUES (2001, 50, 0);
INSERT INTO library_catalog (catalog_number, num_copies, num_out)
VALUES (3001, 5, 0);
INSERT INTO library_catalog (catalog_number, num_copies, num_out)
VALUES (3002, 5, 1);
- Теперь обновим таблицу с помощью переменных PL/SQL.
```

```
UPDATE library_catalog
SET checked_out = v_StudentList1
WHERE catalog_number = 1000;
UPDATE library_catalog
SET checked_out = v_StudentList2
WHERE catalog_number = 1001;
UPDATE library_catalog
SET checked_out = v_StudentList3
WHERE catalog_number = 1002;
- А последнюю строку модифицируем, используя новую переменную.
UPDATE library_catalog
SET checked_out = StudentList(10009)
WHERE catalog_nunber = 3002;
END;
```

DELETE

С помощью оператора DELETE можно удалить строку, содержащую сборную конструкцию, обычным образом, как проиллюстрировано в следующем примере:

```
DELETE FROM library_catalog
WHERE catalog_number = 3001;
```

SELECT

Сборные конструкции, как и любой другой тип базы данных, считываются в переменные PL/SQL оператором SELECT, после чего с ними можно работать с помощью процедурных операторов.

На примере процедуры PrintRequired, которая распечатывает книги, обязательные для данной группы, показано, как SELECT помещает хранимый изменяемый массив в переменную PL/SQL, где с ним можно работать дальше:

```
CREATE OR REPLACE PROCEDURE PrintRequired(
p_Department IN classjnaterial.department%TYPE,
p_Course IN class_material.course%TYPE) IS
v_Books classjnaterial.required_reading%TYPE;
v_Title books.title%TYPE;
BEGIN
-- Извлечем весь изменяемый массив (типа BookList).
SELECT required_reading
INTO v_Books
FROM class_material
WHERE department = p_Department
AND course = p_Course;
DBMS_OUTPUT.PUT(‘Required reading for ' || RTRIM(p_Department));
DBMS_OUTPUT.PUT_LINE(' ' || p_Course || ' : ' ) ;
- В цикле распечатаем каждую строку таблицы.
FOR v_Index IN 1 .. v_Books.COUNT LOOP
SELECT title
INTO v_Title
FROM books
WHERE catalog_number = v_Books(v_Index);
DBMS_OUTPUT.PUT_LINE(
' ' II v_Books(v_Index) || ' : ' || v_Title);
END LOOP;
END PrintRequired;
```

Когда вложенная таблица считывается в переменную PL/SQL, ей присваиваются ключи со значениями от 1 и до числа, равного количеству элементов таблицы. Значение этого числа можно узнать с помощью метода COUNT.

С вложенными таблицами, хранимыми в базе данных, можно работать только в SQL, но не напрямую в PL/SQL, и, как следствие, значения ключей при этом не записываются. Ключи вложенной таблицы при ее считывании из базы данных нумеруются заново по порядку, начиная с 1. Таким образом, при вводе в базу данных вложенной таблицы с непоследовательными ключами те изменяются.

Работа с отдельными элементами сборных конструкций

В рассмотренных выше примерах сборные конструкции модифицировались целиком, однако можно манипулировать и отдельными их элементами, применяя для этого операции PL/SQL и SQL.

Работа в PL/SQL

В PL/SQL обращение осуществляется обычным образом.

SQL-операции TABLE

Работать с элементами хранимых вложенных таблиц можно непосредственно в SQL с помощью оператора TABLE. При использовании этого оператора не нужно извлекать вложенную таблицу в переменную PL/SQL, выполнять над ней действия, а затем обновлять в базе данных.

Однако с элементами изменяемых массивов нельзя работать непосредственно в SQL — ими манипулируют с помощью PL/SQL.

TABLE определяется следующим образом:

TABLE(подзапрос)

где подзапрос — запрос, возвращающий столбец вложенной таблицы.

Пример:

```
PROCEDURE PrintCheckedOut(
p_CatalogNumber IN library_catalog.catalog_number%TYPE) IS
v_StudentList StudentList;
v_Student students%ROWTYPE;
v_Book books%ROWTYPE;
v_FoundOne BOOLEAN := FALSE;
CURSOR c_CheckedOut IS
SELECT column_value ID
FROM TABLE(SELECT checked_out
FROM library_catalog
WHERE catalog_number = p_CatalogNumber);
BEGIN
SELECT *
INTO v_Book
FROM books
WHERE catalog_number = p_CatalogNumber;
DBMS_OUTPUT.PUT_LINE(
'Students who have ' || v_Book.catalog_number || ': ' ||
v_Book.title || ' checked out: ');
- В цикле просмотрим вложенную таблицу и распечатаем имена и фамилии студентов.
FOR v_Rec IN c_CheckedOut LOOP
v_FoundOne := TRUE;
SELECT *
INTO v_Student
FROM students
WHERE ID = v_Rec.ID;
```

```
DBMS_OUTPUT.PUT_LINE(' ' || v_Student.first_name || ' ' ||
v_Student.last_name);
END LOOP;
IF NOT v FoundOne THEN
DBMS_OUTPUT.PUT_LINE(' None');
END IF;
END PrintCheckedOut;
```

Элементами хранимого изменяемого массива нельзя манипулировать с помощью инструкций DML (в отличие от вложенных таблиц), однако изменяемый массив можно запрашивать с помощью оператора TABLE. В этом случае TABLE извлекает столбец изменяемого массива и возвращает его элементы, как если бы изменяемый массив сам был отдельной одностолбцовой таблицей. Имя столбца — column_value. Например, можно запросить class_material следующим образом:

```
SELECT department, course, column_value
FROM class_material, TABLE(required_reading);
```

Методы сборных конструкций

Вложенные таблицы и изменяемые массивы — это объектные типы, и поэтому для них определяется ряд методов. Индексные таблицы обладают атрибутами. Методы и атрибуты сборных конструкций вызываются следующим образом:

эземпляр_сборной_конструкции.метод_или_атрибут

где экземпляр_сборной_конструкции — это переменная сборной конструкции (не имя типа), а метод_или_атрибут — один из методов или атрибутов. Методы сборных конструкций можно вызывать только из процедурных операторов, но не из SQL-операторов.

Условимся, что во всех примерах, рассматриваемых ниже, сделаны следующие объявления:

```
CREATE OR REPLACE TYPE NumTab AS TABLE OF NUMBER;
CREATE OR REPLACE TYPE NumVar AS VARRAY(25) OF NUMBER;
CREATE OR REPLACE PACKAGE IndexBy AS
TYPE NumTab IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
END IndexBy;
```

EXISTS

Метод EXISTS позволяет определить, существует ли реально элемент сборной конструкции, на который производится ссылка. Синтаксис этого метода:

EXISTS(n)

где n — целочисленное выражение. Если элемент, указываемый n, существует, возвращается TRUE (даже если элемент является NULL). Если же значение n лежит вне заданного диапазона, то EXISTS не порождает исключительную ситуацию, а возвращает FALSE.

Допустим для индексных таблиц, вложенных таблиц, изменяемых массивов.

COUNT

Метод COUNT возвращает текущее количество элементов сборной конструкции в виде целого числа. COUNT не имеет аргументов и применяется везде, где разрешены целочисленные выражения.

Для изменяемых массивов метод COUNT эквивалентен методу LAST, так как элементы изменяемых массивов удалять нельзя. Однако из вложенной таблицы можно удалять элементы, поэтому для нее методы COUNT и LAST могут работать по-разному. Метод COUNT наиболее эффективен при выборе вложенной таблицы в базе данных, поскольку в тот момент количество элементов таблицы неизвестно.

Допустим для индексных таблиц, вложенных таблиц, изменяемых массивов.

LIMIT

Метод LIMIT возвращает текущее максимальное число элементов сборной конструкции. Для вложенных таблиц максимальный размер не задается, поэтому при применении к вложенным таблицам LIMIT всегда возвращает NULL. К индексным таблицам LIMIT применять нельзя.

Допустим для вложенных таблиц, изменяемых массивов.

FIRST и LAST

Метод FIRST возвращает индекс первого элемента сборной конструкции, а метод LAST — индекс последнего элемента. Для изменяемого массива FIRST всегда возвращает 1, а LAST — значение COUNT, так как массив является плотным и его элементы удалять нельзя. Методы FIRST и LAST используются совместно с методами NEXT и PRIOR для циклического просмотра сборных конструкций.

NEXT и PRIOR

Методы NEXT и PRIOR служат для увеличения или уменьшения ключа сборных конструкций. Синтаксис их таков:

NEXT(n)

PRIOR(n)

где n— целочисленное выражение. NEXT(n) возвращает ключ элемента, следующего сразу же за элементом n, а PRIOR(n) — ключ элемента, непосредственно предшествующего элементу n. Если предшествующего или следующего элемента нет, PRIOR или NEXT возвращает NULL.

Методы FIRST, LAST, NEXT и PRIOR аналогичным образом работают с изменяемыми массивами и индексными таблицами.

Допустим для индексных таблиц, вложенных таблиц, изменяемых массивов.

EXTEND

Метод EXTEND служит для добавления элементов в конец вложенной таблицы или изменяемого массива. EXTEND имеет три формы:

EXTEND

EXTEND(n)

EXTEND(n, i)

EXTEND без аргументов добавляет элемент NULL в конец сборной конструкции с индексом LAST + 1. EXTEND (n) добавляет в конец таблицы n элементов NULL, а EXTEND(n,z) добавляет в конец таблицы n копий элемента г. Если сборная конструкция была создана с ограничением NOT NULL, то можно применять только последнюю форму, так как при этом NULL- элементы не добавляются.

Для вложенных таблиц максимальный размер не устанавливается явно, поэтому можно вызывать EXTEND со сколь угодно большим n. Изменяемый же массив можно расширять только до размера, объявленного максимальным, поэтому значение n не должно превышать LIMIT—COUNT.

Допустим для вложенных таблиц, изменяемых массивов.

TRIM

TRIM служит для удаления элементов в конце вложенной таблицы или изменяемого массива. Этот метод имеет две формы:

TRIM,

TRIM(n)

Без аргументов TRIM удаляет один последний элемент сборной конструкции. Во втором случае удаляются n элементов. Если n больше COUNT, то порождается исключительная ситуация SUBSCRIPT_BEYOND_COUNT.

После работы метода TRIM значение COUNT уменьшается, так как TRIM удаляет элементы конструкции.

Допустим для вложенных таблиц, изменяемых массивов.

DELETE

Метод DELETE удаляет 1 или более элементов из индексной или вложенной таблицы. DELETE не оказывает воздействия на изменяемые массивы, так как их размер фиксирован (вообще, вызывать DELETE для изменяемого массива запрещено). Метод DELETE имеет три формы:

DELETE

DELETE(n)

DELETE(m,n)

DELETE без аргументов удаляет всю таблицу. DELETE(n) удаляет элемент, имеющий индекс n, а DELETE(m,n) — все элементы, находящиеся между элементами с индексами m и n. После работы метода DELETE значение COUNT уменьшается, отражая новый размер таблицы. Если удаляемый элемент не существует, DELETE не устанавливает никакой исключительной ситуации, а пропускает этот элемент.

Допустим для индексных таблиц, вложенных таблиц.

Триггеры

Типы триггеров

Триггеры похожи на процедуры и функции тем, что также являются именованными блоками PL/SQL и имеют раздел объявлений, выполняемый раздел и раздел обработки исключительных ситуаций. Подобно пакетам, триггеры хранятся как автономные объекты в базе данных и не могут храниться локально в блоке или пакете. Процедура вызывается явным образом из другого блока, при вызове ей могут передаваться различные аргументы. Триггер же выполняется неявно всякий раз, когда происходит запускающее его событие, и триггер не имеет аргументов. Акт выполнения триггера называется его активизацией (firing). Событием, запускающим триггер, является операция DML (INSERT, UPDATE или DELETE), выполняемая над таблицей или представлением базы данных, системное событие (например на запуск или останов базы данных), а также определенные виды операций DDL.

Триггеры можно использовать:

• Для реализации сложных ограничений целостности данных, которые невозможно реализовать через декларативные ограничения, устанавливаемые при создании таблицы.

• Для контроля за информацией, хранимой в таблице, посредством регистрации вносимых изменений и пользователей, производящих эти изменения.

• Для автоматического оповещения других программ о том, что необходимо делать в случае изменения информации, содержащейся в таблице.

• Для публикации информации о различных событиях в среде "публикация- подписка".

Триггеры делятся на три основных типа: триггеры DML, триггеры замещения и системные триггеры.

Создание триггеров

Вне зависимости от типа все триггеры создаются одинаково. Общий синтаксис создания триггера таков:

CREATE [OR REPLACE] TRIGGER имя_триггера

{BEFORE | AFTER | INSTEAD OF} активизирующее_событие

ссылочное_предложение

[WHEN условие_срабатывания]

[FOR EACH ROW]

тело_триггера;

где имя_триггера — это имя триггера, активизирующее_событие указывает событие, которое запускает триггер (может содержать конкретную таблицу или представление), а тело_триггера — основной программный текст триггера. Ссылочное_предложение используется для ссылки на данные в модифицируемой в конкретный момент строке с помощью другого имени. Если присутствует условие_срабатывания. в конструкции WHEN

(когда), то оно оценивается первым. Тело триггера выполняется только в том случае, если это условие истинно.

Создание триггеров DML

Триггер DML активизируется операцией INSERT, UPDATE или DELETE, выполняемой над таблицей базы данных. Триггеры могут активизироваться до (BEFORE) или после (AFTER) операции и действовать на уровне строки или оператора. Тип триггера определяется комбинацией этих факторов. Существует 12 возможных видов: 3 оператора x 2 момента времени x 2 уровня. Ниже приведены примеры правильных триггеров DML:

- До выполнения операции обновления на операторном уровне
- После выполнения операции ввода на уровне строк
- До выполнения операции удаления на уровне строк

Триггер может активизироваться несколькими типами операторов DML, выполняемых над конкретной таблицей: например, INSERT и UPDATE. Код триггера выполняется вместе с активизирующим оператором как часть одной транзакции.

Для таблицы можно создать любое число триггеров каждого вида, в том числе несколько триггеров определенного DML-типа. Например, можно описать два операторных триггера AFTER DELETE. Триггеры одного и того же типа будут срабатывать по очереди.

Предположим, что требуется отслеживать статистические показатели, касающиеся различных профилирующих дисциплин студентов, в том числе количество зарегистрированных студентов и общее число полученных зачетов. Результаты будут храниться в таблице major_stats:

```
CREATE TABLE major_stats (
major VARCHAR2(30)
total_credits NUMBER,
total_students NUMBER);
```

Чтобы информация в таблице major_stats была самой свежей, создадим триггер для таблицы students, который будет обновлять major_stats всякий раз при изменении students. Назовем этот триггер UpdateMajorStats. Он будет срабатывать после выполнения любой операции DML над students.

Тело триггера обращается к таблице students с запросом и обновляет статистические показатели таблицы major_stats свежей информацией:

```
CREATE OR REPLACE TRIGGER UpdateMajorStats
/* Обновляет таблицу major_stats, отслеживая все
изменения, вносимые в таблицу students. */
AFTER INSERT OR DELETE OR UPDATE ON students
DECLARE
CURSOR c_Statistics IS
SELECT major, COUNT(*) total_students,
SUM(current_credits) total_credits
FROM students
GROUP BY major;
BEGIN
/* Сначала удалим информацию из major_stats, очистив статистические данные.
Это необходимо для учета удаления всех студентов данного профиля. */
DELETE FROM major_stats;
/* Теперь в цикле просмотрим информацию по каждой дисциплине и
введем соответствующую строку в major_stats. */
FOR v_StatsRecord in c_Statistics LOOP
INSERT INTO major_stats (major, total_credits, total_students)
VALUES (v_StatsRecord.major, v_StatsRecord.total_credits,
v_StatsRecord.total_students);
```

END LOOP;
END UpdateMajorStats;

Операторный триггер может активизироваться операторами нескольких видов. Например, UpdateMajorStats срабатывает на операторы INSERT, UPDATE и DELETE. Активизирующее событие указывает одну или несколько операций DML, вызывающих выполнение триггера.

Если триггер является строковым, то он активизируется один раз для каждой из строк, на которые воздействует оператор, вызывающий срабатывание триггера. Если триггер является операторным, то он активизируется один раз до или после оператора. Строковые триггеры идентифицируются предложением FOR EACH ROW (для каждой строки) в описании триггера.

В активизирующем событии триггера DML указывается имя таблицы (и столбца), для которой должен срабатывать триггер.

Порядок активизации триггеров DML

Триггеры активизируются при выполнении оператора DML. Алгоритм выполнения оператора DML таков:

1. Выполняются операторные триггеры BEFORE (при их наличии).
2. Для каждой строки, на которую воздействует оператор:
   A. Выполняются строковые триггеры BEFORE (при их наличии).
   B. Выполняется собственно оператор.
   C. Выполняются строковые триггеры AFTER (при их наличии).
3. Выполняются операторные триггеры AFTER (при их наличии).

Порядок, в котором активизируются триггеры одного вида, не определен.

Идентификаторы корреляции в строковых триггерах

Строковый триггер запускается один раз для каждой строки, обрабатываемой активизирующим оператором. Внутри триггера можно обращаться к данным строки, обрабатываемой в данный момент. Для этого служат два идентификатора корреляции — :old и :new. Идентификатор корреляции (correlation identifier) — это переменная привязки PL/SQL особого рода.

Двоеточие перед идентификатором указывает на то, что это переменные привязки (подобны базовым переменным, используемым во встроенном PL/SQL), а не обычные переменные PL/ SQL. Компилятор PL/ SQL рассматривает их как записи типа

активизирующая_таблица%ROWTYPE,

где активизирующая_таблица — это таблица, для которой создан триггер.

Следовательно, ссылка типа

:new.поле

будет достоверна, если только поле является полем активизирующей таблицы. Хотя синтаксически они рассматриваются в качестве записей, фактически эти идентификаторы записями не являются. Именно поэтому их называют псевдозаписями. Операции, применимые к записям, не могут быть выполнены над :new и :old. Кроме того, :old и :new нельзя передавать процедурам или функциям, принимающим аргументы типа активширующая_таблица%ROWTYPE.

Псевдозапись :old не определена для операторов INSERT, а для операторов DELETE не определена псевдозапись :new. В случае использования :old в операторе INSERT или :new в операторе DELETE компилятор PL/SQL не будет генерировать ошибку, но значения полей обеих записей будут NULL.

Одно из полезных свойств :new — когда выполнение оператора завершается, используются те значения, которые содержатся в :new.

Вообще говоря, :new модифицируется только в строковых триггерах BEFORE; :old никогда не модифицируется, а лишь считывается.

Псевдозаписи :new и :old разрешается использовать только в строковых триггерах. Если указать какую-либо из них в операторном триггере, будет выдана ошибка компиляции. Поскольку операторный триггер выполняется лишь однажды (даже в том случае, когда в

операторе обрабатывается несколько строк), псевдозаписи :old и :new не имеют никакого смысла. Действительно, на какую из строк будет ссылаться каждая из них?

Конструкция REFERENCING При желании можно воспользоваться конструкцией REFERENCING и указать другие имена для : old и : new. Эта конструкция размещается после активизирующего события, перед условием

WHEN:

REFERENCING [OLD AS старое_имя] [NEW AS новое_имя]

В теле триггера вместо :old и :new можно использовать :старое_имя и :новое_имя. Отметим, что в предложении REFERENCING идентификаторы указываются без двоеточия. Пример

```
CREATE OR REPLACE TRIGGER GenerateStudentID
BEFORE INSERT OR UPDATE ON students
REFERENCING new AS new _student
FOR EACH ROW
BEGIN
/* Заполним поле ID таблицы students следующим значением из student_sequence. Поскольку ID - это столбец таблицы students, :new_student.ID является допустимой ссылкой. */
SELECT student_sequence.NEXTVAL
INTO :new_student.ID
FROM dual;
END GenerateStudentID;
```

Предложение WHEN

Предложение WHEN можно использовать только для строковых триггеров. При наличии WHEN тело триггера будет выполняться только для тех строк, которые соответствуют условию, указанному в WHEN. Общий вид предложения WHEN таков:

WHEN условие_триггера

где условие_триггера является логическим выражением, которое проверяется для каждой строки. В условии можно ссылаться на записи : new и : old, но двоеточие в данном случае не применяется. Двоеточие можно указывать только в теле триггера. Например, тело триггера CheckCredits выполняется, если текущее число зачетов, полученных студентом, превышает 20:

```
CREATE OR REPLACE TRIGGER CheckCredits
BEFORE INSERT OR UPDATE OF current_credits ON students
FOR EACH ROW
WHEN (new.current_credits> 20)
BEGIN
/* Тело триггера */
END;
```

Триггер CheckCredits можно также написать следующим образом:

```
CREATE OR REPLACE TRIGGER CheckCredits
BEFORE INSERT OR UPDATE OF current_credits ON Students
FOR EACH ROW
BEGIN
IF :new.current_credits > 20 THEN
/* Тело триггера */
END IF;
END;
```

Триггерные предикаты: INSERTING, UPDATING и DELETING

Предположим, некоторый триггер является триггером INSERT, UPDATE и DELETE. Внутри триггера такого типа (который срабатывает на различные виды операторов DML) можно использовать три логические функции, определяющие тип выполняемой операции. Это логические функции (предикаты) INSERTING, UPDATING и DELETING.

Создание замещающих триггеров

В отличие от триггеров DML, срабатывающих как дополнение к операции INSERT, UPDATE или DELETE (или до, или после них), замещающие триггеры активизируются вместо операций DML. К тому же замещающие триггеры создаются только для представлений, в то время как триггеры DML — для таблиц. Замещающие триггеры используются в двух случаях:

- Для того чтобы сделать представление модифицируемым, если иначе это сделать нельзя.
- Для модификации столбцов в столбце вложенной таблицы представления.

Триггеры замещения должны быть строковыми триггерами. Для примера рассмотрим представление classes_rooms:

```
CREATE OR REPLACE VIEW classes_rooms AS
SELECT department, course, building, room_number
FROM rooms, classes
WHERE rooms.room_id = classes.room_id;
```

Ввести информацию непосредственно в это представление нельзя, так как это соединение двух таблиц, и при вводе необходимо модифицировать обе таблицы, как показывает следующий пример:

```
INSERT INTO classes_rooms (department, course, building, room_number)
VALUES ('MUS', 100, 'Music Building', 200);
INSERT INTO classes_rooms (department, course, building, room_number)

ERROR at line 1:
ORA-01776: cannot modify more than one base table through a join view
```

Однако можно создать триггер замещения и с его помощью выполнить обновление базовых таблиц:

```
CREATE TRIGGER ClassesRoomsInsert
INSTEAD OF INSERT ON classes_rooms
DECLARE
v_roomID rooms.room_id%TYPE;
BEGIN
- Сначала определим идентификатор аудитории.
SELECT room_id
INTO v_roomID
FROM rooms
WHERE building = :new.building
AND room_number = :new.room_number;
- А теперь обновим группу.
UPDATE CLASSES
SET room id = v roomlD
WHERE department = :new.department
AND course = :new.course;
END ClassesRoomsInsert;
```

С помощью триггера ClassesRoomsInsert оператор INSERT выполняется успешно и делает именно то, что нужно.

Создание системных триггеров

Как было показано выше, триггеры DML и триггеры замещения срабатывают на события DML (или вместо них), а именно на операторы INSERT, UPDATE и DELETE. Системные же триггеры активизируются событиями двух видов: DDL и базы данных. К событиям DDL относятся операторы CREATE, ALTER и DROP, а к событиям базы данных — запуск/останов сервера, регистрация/отключение пользователя и ошибка сервера. Синтаксис создания системного триггера:

CREATE [OR REPLACE] TRIGGER [схема.]имя_триггера
{BEFORE | AFTER}
[список_событий_ddl \ список_событий_базы_данных}
ON {DATABASE | [схема.]SCHEMA}
[условие_ WHEN]
тело_триггера;

где список_событий_ddl — одно или несколько событий DDL (разделенных ключевым словом OR (или)), а список_событий_базы_данных — одно или несколько событий базы данных (также разделенных ключевым словом OR).

В следующей таблице описаны события DDL и базы данных вместе с указанием допустимого момента их обработки (BEFORE или AFTER). Системные триггеры замещения создавать нельзя. Для оператора TRUNCATE не предусмотрено событие базы данных.

Событие	Допустимое время выполнения	Описание
STARTUP	AFTER	Активизируется при запуске экземпляра базы данных.
SHUTDOWN	BEFORE	Активизируется при остановке экземпляра базы данных. Это событие не активизирует триггер, если база данных останавливается аварийно.
SERVERERROR	AFTER	Активизируется при возникновении ошибки.
LOGON	AFTER	Активизируется после успешного соединения пользователя с базой данных.
LOGOFF	BEFORE	Активизируется перед отключением пользователя.
CREATE	BEFORE, AFTER	Активизируется до или после создания объекта схемы.
DROP	BEFORE, AFTER	Активизируется до или после удаления объекта схемы.
ALTER	BEFORE, AFTER	Активизируется до или после изменения объекта схемы.

Для создания системного триггера необходимо иметь системную привилегию ADMINISTER DATABASE TRIGGER

Предположим, что необходимо регистрировать моменты создания объектов словаря данных. Это можно сделать, создав следующую таблицу:

CREATE TABLE ddl_creations (
user_id VARCHAR2(30),
object_type VARCHAR2(20),
object_name VARCHAR2(30),
object_owner VARCHAR2(30),
creation date DATE);

После этого можно создать системный триггер для регистрации нужных сведений. Триггер LogCreations регистрирует в таблице ddl_creations сведения о только что созданных объектах после каждой операции CREATE в текущей схеме.

```
CREATE OR REPLACE TRIGGER LogCreations
AFTER CREATE ON SCHEMA
BEGIN
INSERT INTO ddl_creations (user_id, object_type, object_name,
object_owner, creation_date)
VALUES (USER, SYS.DICTIONARY_OBJ_TYPE, SYS.DICTIONARY_OBJ_NAME,
SYS.DICTIONARY_OBJ_OWNER, SYSDATE);
END LogCreations;
```

Триггеры базы данных и триггеры схемы

Системный триггер можно определить на уровне базы данных или на уровне схемы. Триггер базы данных срабатывает каждый раз при наступлении активизирующего события, а триггер схемы — только когда активизирующее событие происходит в указанной схеме. Уровень системного триггера определяют ключевые слова DATABASE и SCHEMA. Если с ключевым словом SCHEMA не указана схема, то по умолчанию используется та, которой принадлежит триггер.

Атрибутные функции событий

Существует ряд атрибутных функций, которые разрешается использовать в системных триггерах. Подобно триггерным предикатам (INSERTING, UPDATING и DELETING), они позволяют получать информацию об активизирующем событии в теле триггера. Хотя эти функции можно вызывать и из других блоков PL/SQL (не только в теле системного триггера), возвращаемый ими результат не всегда будет достоверным.

Например:

SYSEVENT – возвращает системное событие, активизировавшее триггер;

DATAVASE_NAME – возвращает имя текущей базы данных.

Имена триггеров

Пространство имен триггеров отличается от пространств имен других подпрограмм. Пространством имен (namespace) называется набор идентификаторов, разрешенных для применения в качестве имен объектов. Для процедур, модулей и таблиц применяется одно и то же пространство имен. Это означает, что в пределах одной схемы базы данных все объекты, использующие одно и то же пространство имен, должны иметь уникальные имена. Например, запрещается давать одинаковые имена процедуре и модулю.

Для триггеров же определено собственное пространство имен, т.е. триггер может иметь то же имя, что и какая-либо таблица или процедура. Однако в пределах одной схемы конкретное имя может быть дано только одному триггеру.

Хотя не запрещается применять для таблицы и триггера одинаковые имена, делать это не рекомендуется. Лучше дать каждому триггеру уникальное имя, показывающее, какие функции он выполняет или для какой таблицы он создан.

Удаление и запрещение триггеров

Триггеры, как и процедуры и модули, можно удалять. Синтаксис команды удаления триггера таков:

```
DROP TRIGGER имя_триггера;
```

где имя_триггера — имя удаляемого триггера. При этом триггер удаляется из словаря данных. В операторе создания триггера можно указывать ключевые слова OR REPLACE, как это делается для подпрограмм. В этом случае если триггер существует, то сначала он удаляется.

Однако в отличие от процедур и функций можно, не удаляя триггер, запретить (DISABLE) его использование. Запрещенный триггер находится в словаре данных, но никогда не активизируется. Для запрещения триггера применяется оператор ALTER TRIGGER:

```
ALTER TRIGGER имя_триггера {DISABLE | ENABLE};
```

При создании триггера его использование разрешено (ENABLE) по умолчанию. С помощью оператора ALTER TRIGGER можно запретить, а затем повторно разрешить любой триггер.

Кроме того, при помощи команды ALTER TABLE можно разрешить или запретить использование всех триггеров определенной таблицы, если добавить конструкцию ENABLE ALL TRIGGERS (разрешить все триггеры) или DISABLE ALL TRIGGERS (запретить все триггеры). Например:

```
ALTER TABLE students
ENABLE ALL TRIGGERS;
```

Блокирование и одновременный доступ

Несколько замечаний из многолетнего опыта разработки Тома Кайта:

• успех или неудача разработки приложения базы данных (приложения, зависящего от базы данных) определяется тем, как оно использует базу данных;

• в команде разработчиков должно быть ядро "программистов базы данных", обеспечивающих согласованность логики работы с базой данных и настройку производительности системы.

Эти утверждения могут показаться очевидными, однако слишком многие используют СУБД как "черный ящик", о деталях устройства которого знать необязательно. Они могут использовать генератор SQL, позволяющий не затруднять себя изучением языка SQL. Возможно, они решат использовать базу данных как обычный файл с возможностью чтения записей по ключу. Однако, подобного рода соображения почти наверняка приводят к неправильным выводам — работать, не понимая устройства СУБД, просто нельзя.

В этой главе мы рассмотрим, почему необходимо знать устройство СУБД.

Наиболее типичной причиной неудачи многих проектов разработки ПО является нехватка практических знаний по используемой СУБД — элементарное непонимание основ работы используемого инструментального средства. Подход по принципу "черного ящика" требует осознанного решения: оградить разработчиков от СУБД. Их заставляют не вникать ни в какие особенности ее функционирования.

Вот типичный сценарий такого рода разработки.

• Разработчики были полностью обучены графической среде разработки или соответствующему языку программирования (например, Java), использованных для создания клиентской части приложения. Во многих случаях они обучались несколько недель, если не месяцев.

• Команда разработчиков ни одного часа не изучала СУБД Oracle и не имела никакого опыта работы с ней. Многие разработчики вообще впервые сталкивались с СУБД.

• В результате разработчики столкнулись с огромными проблемами, связанными с производительностью, обеспечением целостности данных, зависанием приложений и т.д. (но пользовательский интерфейс выглядел отлично).

Особенности управления одновременным доступом

Одна из основных проблем при разработке многопользовательских приложений баз данных — обеспечить одновременный доступ максимальному количеству пользователей при согласованном чтении и изменении данных каждым из них. Именно это постоянно забывают проверять.

Механизмы блокирования и управления одновременным доступом, позволяющие решить эту проблему, являются ключевыми в любой базе данных. Именно они являются фундаментальным отличием различных СУБД. Приемы, прекрасно работающие в условиях последовательного доступа, работают гораздо хуже при одновременном их применении несколькими сеансами. Если не знать досконально, как в конкретной СУБД реализованы механизмы управления одновременным доступом, то:

• будет нарушена целостность данных;

• приложение будет работать медленнее, чем предусмотрено, даже при небольшом количестве пользователей;

• будет потеряна возможность масштабирования до большого числа пользователей.

При неправильном управлении одновременным доступом будет нарушена целостность данных, поскольку то, что работает отдельно, будет работать не так, как предполагалось, в многопользовательской среде. Приложение будет работать медленнее, поскольку придется ждать доступа к данным. Возможность масштабирования будет потеряна из-за проблем с блокированием и конфликтов блокировок. По мере усложнения запросов к ресурсу ждать придется все дольше и дольше.

Проблемы одновременного доступа выявлять сложнее всего — трудности сопоставимы с отладкой многопотоковой программы. Программа может отлично работать в управляемой, искусственной среде отладчика, но постоянно "слетать" в "реальном мире".

Например, в условиях интенсивных обращений может оказаться, что два потока одновременно изменяют одну и ту же структуру данных. Такого рода ошибки очень сложно выявлять и исправлять. Если приложение тестировалось только в однопользовательском режиме, а затем внедряется в среде с десятками одновременно обращающихся пользователей, вполне вероятно проявление болезненных проблем с одновременным доступом.

Проблемы блокирования

Потерянные изменения.

Потерянное изменение — классическая проблема баз данных. Если коротко, потерянное изменение возникает, когда происходят следующие события (в указанном порядке).

1. Пользователь 1 выбирает (запрашивает) строку данных.
2. Пользователь 2 выбирает ту же строку.
3. Пользователь 1 изменяет строку, обновляет базу данных и фиксирует изменение.
4. Пользователь 2 изменяет строку, обновляет базу данных и фиксирует изменение.

Это называется потерянным изменением, поскольку все сделанные на шаге 3 изменения будут потеряны. Рассмотрим, например, окно редактирования информации о сотруднике, позволяющее изменить адрес, номер рабочего телефона и т.д. Само приложение — очень простое: небольшое окно поиска со списком сотрудников и возможность получить детальную информацию о каждом сотруднике. Проще некуда. Так что пишем приложение, не выполняющее никакого блокирования, — только простые операторы SELECT и UPDATE.

Итак, пользователь (пользователь 1) переходит к окну редактирования, изменяет там адрес, щелкает на кнопке Save и получает подтверждение успешного обновления. Все отлично, кроме того, что, проверяя на следующий день эту запись, чтобы послать сотруднику налоговую декларацию, пользователь 1 увидит в ней старый адрес. Как это могло случиться? К сожалению, очень просто: другой пользователь (пользователь 2) запросил ту же запись за 5 минут до того, как к ней обратился пользователь 1, и у него на экране отображались старые данные. Пользователь 1 запросил данные, изменил их, получил подтверждение изменения и даже выполнил повторный запрос, чтобы увидеть эти изменения. Однако затем пользователь 2 изменил поле номера рабочего телефона и щелкнул на кнопке сохранения, не зная, что переписал старые данные поверх внесенных пользователем 1 изменений адреса! Так может случиться потому, что разработчик приложения предпочел обновлять сразу все столбцы, а не разбираться, какой именно столбец был изменен, и написал программу так, что при изменении одного из полей обновляются все.

Обратите внимание, что для потери изменений пользователям 1 и 2 вовсе не обязательно работать с записью одновременно. Нужно, чтобы они работали с ней примерно в одно и то же время.

Эта проблема баз данных проявляется постоянно, когда разработчики графических интерфейсов, не имеющие достаточного опыта работы с базами данных, получают задание создать приложение для базы данных. Они получают общее представление об операторах SELECT, INSERT, UPDATE и DELETE и начинают писать программы.

Как средства разработки, обеспечивающие защиту (за кадром), так и разработчики, использующие другие средства (явно), должны применять один из двух описанных ниже методов блокирования.

Пессимистическое блокирование

Этот метод блокирования должен использоваться непосредственно перед изменением значения на экране, например, когда пользователь выбирает определенную строку с целью изменения (допустим, щелкая на кнопке в окне). Итак, пользователь запрашивает данные без блокирования:

```
scott@TKYTE816> SELECT EMPNO, ENAME, SAL FROM EMP WHERE DEPTNO = 10;
EMPNO ENAME SAL
7782 CLARK 2450
7839 KING 5000
7934 MILLER 1300
```

В какой-то момент пользователь выбирает строку для потенциального изменения.

Пусть в этом случае он выбрал строку, соответствующую сотруднику MILLER. Наше приложение в этот момент (перед выполнением изменений на экране) выполняет следующую команду:

```
scott@TKYTE816> SELECT EMPNO, ENAME, SAL
2 FROM EMP
3 WHERE EMPNO = :EMPNO
4 AND ENAME = :ENAME
5 AND SAL = :SAL
6 FOR UPDATE NOWAIT
7 /
EMPNO ENAME SAL
7934 MILLER 1300
```

Приложение передает значения для связываемых переменных в соответствии с данными на экране (в нашем случае — 7934, MILLER и 1300) и повторно запрашивает ту же самую строку из базы данных, но в этот раз блокирует ее изменения другими сеансами. Вот почему такой подход называется пессимистическим блокированием. Мы блокируем строку перед попыткой изменения, поскольку сомневается, что она останется неизменной.

Поскольку все таблицы имеют первичный ключ (приведенный выше оператор SELECT выберет не более одной строки, поскольку критерий выбора включает первичный ключ EMPNO), а первичные ключи должны быть неизменны, при выполнении этого оператора возможен один из трех результатов.

• Если данные не изменились, мы получим ту же строку сотрудника MILLER, на этот раз заблокированную от изменения (но не чтения) другими сеансами.

• Если другой сеанс находится в процессе изменения данной строки, мы получим сообщение об ошибке ORA-00054 Resource Busy (ресурс занят). Наш сеанс заблокирован и мы должны ждать, пока другой сеанс не завершит изменения строки.

• Если за период между выборкой данных и попыткой их изменить другой сеанс уже изменил соответствующую строку, мы получим в результате ноль строк. Данные на экране не обновятся. Приложение должно повторно запросить и заблокировать данные, прежде чем разрешить пользователю изменять их, чтобы предотвратить описанную выше ситуацию с потерей изменений. В этом случае, если используется пессимистическое блокирование, когда пользователь 2 пытается изменить поле номера телефона, приложение "поймет", что изменилось поле адреса, и повторно запросит данные. Поэтому пользователь 2 никогда не перезапишет старые данные поверх изменений, внесенных в это поле пользователем 1.

После успешного блокирования строки приложение выполняет требуемые изменения и фиксирует их:

```
scott@TKYTE816> UPDATE EMP
```

```
2 SET ENAME = :ENAME, SAL = :SAL
3 WHERE EMPNO = :EMPNO;
1 row updated.
scott@TKYTE816> commit;
Commit complete.
```

Теперь мы абсолютно безопасно изменили соответствующую строку. Мы не могли стереть изменения, сделанные в другом сеансе, поскольку проверили, что данные не изменились с момента первоначального чтения до момента блокирования.

Оптимистическое блокирование

Второй метод, который называют оптимистическим блокированием, состоит в том, чтобы сохранять старое и новое значения в приложении и использовать их при изменении следующим образом:

```
Update table
Set column1 = :new_column1, column2 = :new_column2, ....
Where column1 = :old_column1
And column2 = :old_column2
```

Здесь мы оптимистически надеемся, что данные не изменились. Если в результате изменена одна строка, значит, нам повезло: данные не изменились с момента считывания. Если изменено ноль строк, мы проиграли — кто-то уже изменил данные и необходимо решить, что делать, чтобы это изменение не потерять.

Стоит заметить, что и в этом случае тоже можно использовать оператор SELECT FOR UPDATE NOWAIT. Представленный выше оператор UPDATE позволяет избежать потери изменений, но может приводить к блокированию, "зависая" в ожидании завершения изменения строки другим сеансом. Если все приложения используют оптимистическое блокирование, то применение простых операторов UPDATE вполне допустимо, поскольку строки блокируются на очень короткое время выполнения и фиксации изменений. Однако если некоторые приложения используют пессимистическое блокирование, удерживая блокировки строк достаточно долго, имеет смысл выполнять оператор SELECT FOR UPDATE NOWAIT непосредственно перед оператором UPDATE, чтобы избежать блокирования другим сеансом.

Итак, какой же метод лучше? Пессимистическое блокирование очень хорошо работает в Oracle (но вряд ли так же хорошо подходит для других СУБД) и имеет много преимуществ по сравнению с оптимистическим.

При использовании пессимистического блокирования пользователь может быть уверен, что изменяемые им на экране данные сейчас ему "принадлежат" — он получил запись в свое распоряжение, и никто другой не может ее изменять. Можно возразить, что, блокируя строку до изменения, вы лишаете к ней доступа других пользователей и, тем самым, существенно снижаете масштабируемость приложения. Но обновлять строку в каждый момент времени сможет только один пользователь (если мы не хотим потерять изменения). Если сначала заблокировать строку, а затем изменять ее, пользователю будет удобнее работать. Если же пытаться изменить, не заблокировав заранее, пользователь может напрасно потерять время и силы на изменения, чтобы в конечном итоге получить сообщение: "Извините, данные изменились, попробуйте еще раз". Чтобы ограничить время блокирования строки перед изменением, можно снимать блокировку в приложении, если пользователь занялся чем-то другим и некоторое время не использует строку, или использовать профили ресурсов (Resource Profiles) в базе данных для отключения простаивающих сеансов.

Более того, блокирование строки в Oracle не мешает ее читать, как в других СУБД; блокирование строки не мешает обычной работе с базой данных. Все это исключительно благодаря соответствующей реализации механизмов одновременного доступа и блокирования в Oracle. В других СУБД верно как раз обратное. Если попытаться использовать в них пессимистическое блокирование, ни одно приложение не будет работать. Тот факт, что в этих СУБД блокирование строки не дает возможности выполнять к ней запросы, не позволяет даже

рассматривать подобный подход. Поэтому иногда приходится "забывать" правила, выработанные в процессе работе с одной СУБД, чтобы успешно разрабатывать приложения для другой.

Таким образом, оператор SELECT FOR UPDATE NOWAIT, позволяет:

• проверить, не изменились ли данные с момента их прочтения (для предотвращения потерянного изменения);

• заблокировать строку (предотвращая ее блокирование другим оператором изменения или удаления).

Как уже упоминалось, это можно сделать независимо от принятого подхода — как при пессимистическом, так и при оптимистическом блокировании можно использовать оператор SELECT FOR UPDATE NOWAIT для проверки того, что строка не изменилась. При пессимистическом блокировании этот оператор выполняется в тот момент, когда пользователь выражает намерение изменять данные. При оптимистическом блокировании этот оператор выполняется непосредственно перед изменением данных в базе.

Это не только решает проблемы блокирования в приложении, но и обеспечивает целостность данных.

Реализация блокирования

Блокировки — это механизм обеспечения одновременного доступа. Блокировки используются в базе данных для одновременного доступа к общим ресурсам и обеспечения при этом целостности и согласованности данных.

При отсутствии определенной модели блокирования, предотвращающей одновременное изменение, например, одной строки, многопользовательский доступ к базе данных попросту невозможен. Однако при избыточном или неправильном блокировании одновременный доступ тоже может оказаться невозможным. Если пользователь или сама СУБД блокирует данные без необходимости, то работать одновременно сможет меньшее количество пользователей. Поэтому понимание назначения блокирования и способов его реализации в используемой СУБД принципиально важно для создания корректных и масштабируемых приложений.

Принципиально важно также понимать, что в различных СУБД блокирование реализовано по-своему. В одних — используется блокирование на уровне страниц, в других — на уровне строк; в некоторых реализациях выполняется эскалация блокировок со строчного на страничный уровень, в других — не выполняется; в некоторых СУБД используются блокировки чтения, в других — нет и т.д. Эти небольшие отличия могут перерасти в огромные проблемы, связанные с производительностью, или даже привести к возникновению ошибок в приложениях, если не понимать их особенностей.

Ниже приведены принципы блокирования в СУБД Oracle.

• Oracle блокирует данные на уровне строк и только при изменении. Эскалация блокировок до уровня блока или таблицы никогда не выполняется.

• Сервер Oracle блокирует данные таблицы на уровне строк, но для обеспечения одновременного доступа к различным ресурсам он использует блокировки и на других уровнях. Например, при выполнении хранимой процедуры она блокируется в режиме, который позволяет другим сеансам ее выполнять, запрещая при этом изменять ее.

• Oracle никогда не блокирует данные с целью считывания. При обычном чтении блокировки на строки не устанавливаются.

• Сеанс, записывающий данные, не блокирует сеансы, читающие данные. Это принципиально отличается от практически всех остальных СУБД, в которых операции чтения блокируются операциями записи.

• Сеанс записи данных блокируется, только если другой сеанс записи уже заблокировал строку, которую предполагается изменять. Сеанс считывания данных никогда не блокирует сеанс записи.

Эти факты необходимо учитывать при разработке приложений, однако следует помнить, что эти принципы используются только в Oracle. Разработчик, не понимающий, как

используемая СУБД обеспечивает одновременный доступ, неизбежно столкнется с проблемами целостности данных.

Один из побочных эффектов принятого в СУБД Oracle "неблокирующего" подхода состоит в том, что если действительно необходимо обеспечить доступ к строке не более чем одного пользователя в каждый момент времени, то именно разработчику необходимо предпринять для этого определенные усилия.

В качестве примера рассмотрим программу планирования ресурсов (учебных классов, проекторов и т.д.). Приложение реализовывает бизнес-правило, предотвращающее выделение ресурса более чем одному лицу на любой период времени. То есть, приложение содержит специальный код, который проверяет, что никто из пользователей не затребовал ресурс на тот же период времени. Код обращается к таблице планов и, если в ней не было строк с перекрывающимся временным интервалом, вставляет в нее новую строку. Используемые таблицы:

```
create table resources(resource_name varchar2(25) primary key, . . . );
create table schedules(resource_name varchar2(25) references resources,
start_time date,
end_time date);
```

Прежде чем зарезервировать на определенный период, скажем, учебный класс, приложение выполняет запрос вида:

```
select count(*)
from schedules
where resource_name = :room_name
and (start_time between :new_start_time and :new_end_time
or
end_time between :new_start_time and :new_end_time)
```

Запрос кажется простым и надежным: если возвращено значение 0, учебный класс можно занимать; если возвращено ненулевое значение, значит, учебный класс на этот период уже кем-то занят.

Однако он работает некорректно. При реальной эксплуатации приложения будет возникать ошибка, причем ее будет крайне сложно обнаружить и тем более установить ее причину, — кому-то может даже показаться, что это ошибка СУБД.

С разных компьютеров можно, если делать это одновременно, зарезервировать класс на одно и то же время. Проблема вызвана неблокирующим чтением в Oracle. Ни один из сеансов не блокировал другой.

Оба сеанса просто выполняют представленный выше запрос и применяют алгоритм резервирования ресурса. Они оба могут выполнять запрос, проверяющий занятость ресурса, даже если другой сеанс уже начал изменять таблицу планов (это изменение невидимо для других сеансов до фиксации, то есть до тех пор, когда уже слишком поздно). Поскольку сеансы никогда не пытались изменить одну и ту же строку в таблице планов, они никогда и не блокировали друг друга, вследствие чего бизнес-правило не срабатывало так, как ожидалось.

В данной ситуации разработчику необходим метод реализации данного бизнес-правила в многопользовательской среде, способ, гарантирующий, что в каждый момент времени только один сеанс резервирует данный ресурс. В данном случае решение состоит в программном упорядочении доступа — кроме представленного выше запроса count(*), необходимо сначала выполнить:

```
select * from resources where resource_name = :room_name FOR UPDATE;
```

Мы просто блокируем ресурс (учебный класс), использование которого планируется непосредственно перед его резервированием, до выполнения запроса к таблице планов, выбирающего строки для данного ресурса. Блокируя ресурс, который мы пытаемся зарезервировать, мы гарантируем, что никакой другой сеанс в это же время не изменяет план использования ресурса. Ему придется ждать, пока наша транзакция не будет зафиксирована —

после этого он сможет увидеть сделанное в ней резервирование. Возможность перекрытия планов, таким образом, устранена. Разработчик должен понимать, что в многопользовательской среде иногда необходимо использовать те же приемы, что и при многопотоковом программировании. В данном случае конструкция FOR UPDATE работает как семафор. Она обеспечивает последовательный доступ к конкретной строке в таблице ресурсов, гарантируя, что два сеанса одновременно не резервируют ресурс.

Этот подход обеспечивает высокую степень параллелизма, поскольку резервируемых ресурсов могут быть тысячи, а мы всего лишь гарантируем, что сеансы изменяют конкретный ресурс поочередно. Это один из немногих случаев, когда необходимо блокирование вручную данных, которые не должны изменяться. Требуется уметь распознавать ситуации, когда это необходимо, и, что не менее важно, когда этого делать не нужно. Кроме того, такой прием не блокирует чтение ресурса другими сеансами, как это могло бы произойти в других СУБД, благодаря чему обеспечивается высокая масштабируемость.

Подобные проблемы приводят к масштабным последствиям при переносе приложения с одной СУБД на другую, которых разработчикам сложно избежать. Например, при наличии опыта разработки для другой СУБД, в которой пишущие сеансы блокируют читающих, и наоборот, разработчик может полагаться на это блокирование как защищающее от подобного рода проблем — именно так все и работает во многих СУБД, отличных от Oracle. В Oracle приоритет отдан одновременности доступа, и необходимо учитывать, что в результате все может работать по-другому.

Нужно понимать, как приложение будет работать в многопользовательской среде и что оно будет делать в базе данных.

За исключением некоторых приложений, исключительно читающих из базы данных, создать полностью независимое от СУБД и при этом масштабируемое приложение крайне сложно и даже практически невозможно, не зная особенностей работы всех СУБД.

Например, давайте вернемся к примеру планировщика ресурсов (до добавления конструкции FOR UPDATE). Предположим, это приложение было разработано на СУБД с моделью блокирования/обеспечения одновременного доступа, полностью отличающейся от принятой в Oracle. Сейчас мы увидим, что при переводе приложения с одной СУБД на другую необходимо проверять, работает ли оно корректно в новой среде.

Предположим, что первоначально приложение по планированию ресурсов работало в СУБД, использующей блокирование на уровне страниц и блокировку чтения (чтение блокируется при изменении считываемых данных), и для таблицы SCHEDULES был создан индекс:

```
create index schedules_idx on schedules(resource name, start_time);
```

Предположим также, что бизнес-правило было реализовано с помощью триггера (после выполнения оператора INSERT, но перед фиксацией транзакции мы проверяем, что для указанного временного интервала в базе данных имеется только наша, только что вставленная строка). В системе с блокированием на уровне страниц, из-за изменения страницы индекса по столбцам RESOURCE_NAME и START_TIME, очень вероятно, что транзакции будут выполняться строго последовательно. Система будет выполнять вставки поочередно, поскольку страница индекса блокируется (все близкие значения по полю START_TIME для одного ресурса RESOURCE_NAME будут находиться на той же странице). В такой СУБД с блокированием на уровне страниц наше приложение, вероятно, будет работать нормально, так как перекрытие выделяемых ресурсов будет проверяться последовательно, а не одновременно.

Если просто перенести это приложение в СУБД Oracle, исходя из предположения, что она работает точно так же, можно получить шок. В СУБД Oracle, выполняющей блокирование на уровне строк и не блокирующей чтения, оно окажется некорректным.

Как уже было показано, необходимо использовать конструкцию FOR UPDATE для упорядочения доступа. Без этой конструкции два пользователя могут зарезервировать ресурс на

одно и то же время. Это будет прямым следствием непонимания особенностей работы используемой СУБД в многопользовательской среде.

С подобные проблемы многократно возникают при переносе приложений из СУБД А в СУБД Б. Когда приложение, без проблем работавшее в СУБД А, не работает или работает весьма странно в СУБД Б, сразу же возникает мысль, что "СУБД Б — плохая". Правда, однако, в том, что СУБД Б работает иначе. Ни одна из СУБД не ошибается и не является "плохой" — они просто разные. Знание и понимание особенностей их работы поможет успешно решить подобные проблемы.

Можно сделать следующие важные выводы.

• СУБД — различны. Опыт работы с одной может оказаться полезен в другой, но нужно быть готовым к ряду принципиальных отличий и многим очень мелким.

• Мелкие различия (вроде обработки NULL-значений) могут иметь такое же влияние, как и принципиальные (например, механизм управления одновременным доступом).

• Единственный способ справиться с этими проблемами — знать особенности работы СУБД и уметь реализовать предоставляемые ею возможности.

Взаимные блокировки.

Взаимные блокировки возникают, когда два сеанса удерживают ресурсы, необходимые другому сеансу. Взаимную блокировку легко продемонстрировать на примере базы данных с двумя таблицами, А и В, в каждой из которых по одной строке. Для этого необходимо начать два сеанса (скажем, два сеанса SQL*Plus) и в сеансе А изменить таблицу А. В сеансе Б надо изменить таблицу В. Теперь, если попытаться изменить таблицу А в сеансе Б, он окажется заблокированным, поскольку соответствующая строка уже заблокирована сеансом А. Это еще не взаимная блокировка — сеанс просто заблокирован. Взаимная блокировка еще не возникла, поскольку есть шанс, что сеанс А зафиксирует или откатит транзакцию и после этого сеанс Б продолжит работу.

Если мы вернемся в сеанс А и попытаемся изменить таблицу В, то вызовем взаимную блокировку. Один из сеансов будет выбран сервером в качестве "жертвы", и в нем произойдет откат оператора. Например, может быть отменена попытка сеанса Б изменить таблицу А с выдачей сообщения об ошибке следующего вида:

```
update a set x = x+1
*
ERROR at line 1:
ORA-00060: deadlock detected while waiting for resource
```

Попытка сеанса А изменить таблицу В по-прежнему блокируется — сервер Oracle не откатывает всю транзакцию. Откатывается только один из операторов, ставших причиной возникновения взаимной блокировки. Сеанс Б по-прежнему удерживает блокировку строки в таблице В, а сеанс А терпеливо ждет, пока эта строка станет доступной. Получив сообщение о взаимной блокировке, сеанс Б должен решить, фиксировать ли уже выполненные изменения в таблице В, откатить ли их или продолжить работу и зафиксировать транзакцию позднее. Как только этот сеанс зафиксирует или откатит транзакцию, другие заблокированные сеансы смогут продолжить работу.

Для сервера Oracle взаимная блокировка — настолько редкий, необычный случай, что при каждом ее возникновении создается трассировочный файл.

Очевидно, сервер Oracle воспринимает взаимные блокировки как ошибки приложения, и в большинстве случаев это справедливо. В отличие от многих других реляционных СУБД, взаимные блокировки настолько редко происходят в Oracle, что вполне можно игнорировать их существование. Обычно взаимную блокировку необходимо создавать искусственно.

Как свидетельствует опыт, основной причиной возникновения взаимных блокировок в базах данных Oracle являются неиндексированные внешние ключи. При изменении главной таблицы сервер Oracle полностью блокирует подчиненную таблицу в двух случаях:

• при изменении первичного ключа в главной таблице (что бывает крайне редко, если следовать принятому в реляционных базах данных правилу неизменности первичных ключей) подчиненная таблица блокируется при отсутствии индекса по внешнему ключу;

• при удалении строки в главной таблице подчиненная таблица также полностью блокируется (при отсутствии индекса по внешнему ключу).

Чтобы продемонстрировать первый случай, создадим пару таблиц следующим образом:

```
tkyte@TKYTE816> create table p (x int primary key) ;
Table created.
tkyte@TKYTE816> create table c (y references p) ;
Table created.
tkyte@TKYTE816> insert into p values (1) ;
tkyte@TKYTE816> insert into p values (2) ;
tkyte@TKYTE816> commit;
```

А затем выполним:

```
tkyte@TKYTE816> update p set x = 3 where x = 1;
1 row updated.
```

В результате сеанс заблокировал таблицу C, и никакой другой сеанс не может удалять, вставлять или изменять в ней строки. Повторю еще раз: изменение первичного ключа не приветствуется при работе с реляционными базами данных, так что обычно подобная проблема не возникает. Зато проблема изменения первичного ключа становится актуальной в случае использования средств автоматической генерации SQL-операторов, которые обновляют значения всех столбцов, независимо от того, изменил ли значение пользователь. Например, при создании в Oracle Forms стандартной формы для просмотра и редактирования таблицы по умолчанию генерируется оператор UPDATE, изменяющий все просматриваемые столбцы таблицы. За этим надо внимательно следить при использовании любого средства, автоматически генерирующего SQL-операторы.

Проблемы, связанные с удалением строки в главной таблице, возникают намного чаще. Если удаляется строка в таблице P, то вся подчиненная таблица C оказывается заблокированной, что не позволяет выполнять другие изменения таблицы C в течение всей транзакции (предполагается, конечно, что ни один другой сеанс не изменял таблицу C в момент удаления, иначе оператору удаления пришлось бы ожидать). Именно так возникают проблемы блокирования, в том числе взаимного. Блокирование таблицы C ограничивает возможность одновременной работы с базой данных, — любые изменения в ней становятся невозможными. Кроме того, увеличивается вероятность взаимного блокирования, поскольку сеанс в течение транзакции "владеет" слишком большим объемом данных. Вероятность того, что другой сеанс окажется заблокированным при попытке изменения таблицы C, теперь намного больше. Вследствие этого блокируется множество сеансов, удерживающих определенные ресурсы. Если какой-либо из заблокированных сеансов удерживает ресурс, необходимый исходному, удалившему строку сеансу, возникает взаимная блокировка. Причина взаимной блокировки в данном случае — блокирование исходным сеансом намного большего количества строк, чем реально необходимо.

Проиндексировав столбец Y, мы можем полностью устранить проблему блокирования. Помимо блокирования таблицы неиндексированный внешний ключ может вызывать проблемы в следующих случаях:

• Если установлено ограничение ON DELETE CASCADE и подчиненная таблица не проиндексирована. При удалении строки в главной таблице будет осуществляться полный просмотр строк в подчиненной таблице. Полный просмотр обычно нежелателен, поскольку при удалении большого количества строк из главной таблицы приводит к большим расходам времени и ресурсов.

• При выполнении запроса из главной таблицы в подчиненную.

Итак, когда индекс по внешнему ключу не нужен? В общем случае, когда выполнены следующие условия:

• не удаляются строки из главной таблицы;

• не изменяется значение уникального/первичного ключа главной таблицы (следите за непреднамеренным обновлением первичного ключа используемыми инструментальными средствами!);

• не выполняется соединение главной и подчиненной таблиц.

Если соблюдены все три условия, индекс можно не создавать: он не нужен. Если выполняется одно из перечисленных действий, помните о последствиях. Это один из немногих случаев, когда сервер Oracle "избыточно" блокирует данные.

Эскалация блокирования.

Когда происходит эскалация блокирования, система увеличивает размер блокируемых объектов. Примером может служить блокирование системой всей таблицы вместо 100 отдельных ее строк. В результате одной блокировкой удерживается намного больше данных, чем перед эскалацией. Эскалация блокирования часто используется в СУБД, когда требуется избежать лишнего расходования ресурсов.

В СУБД Oracle никогда не применяется эскалация блокирования, однако выполняется преобразование блокировок (lock conversion) или распространение блокировок (lock promotion). Эти термины часто путают с эскалацией блокирования.

Термины "преобразование блокировок" и "распространение блокировок" — синонимы.

В контексте Oracle обычно говорят о преобразовании.

Берется блокировка самого низкого из возможных уровней (наименее ограничивающая блокировка) и преобразуется к более высокому (ограничивающему) уровню. Например, при выборе строки из таблицы с конструкцией FOR UPDATE будет создано две блокировки. Одна из них устанавливается на выбранную строку (или строки); это — исключительная блокировка: ни один сеанс уже не сможет заблокировать соответствующие строки в исключительном режиме. Другая блокировка, ROW SHARE TABLE (совместное блокирование строк таблицы), устанавливается на соответствующую таблицу.

Это предотвратит исключительную блокировку таблицы другими сеансами и, следовательно, возможность изменения, например, структуры таблицы. Все остальные операторы смогут работать с таблицей. Другой сеанс может даже сделать таблицу доступной только для чтения с помощью оператора LOCK TABLE X IN SHARE MODE, предотвратив тем самым ее изменение. Однако этот другой сеанс не должен иметь права предотвращать изменения, которые уже происходят. Поэтому, как только будет выполнена команда фактического изменения строки, сервер Oracle преобразует блокировку ROW SHARE TABLE в более ограничивающую блокировку ROW EXCLUSIVE TABLE, и изменение будет выполнено. Такое преобразование блокировок происходит само собой независимо от приложений.

Эскалация блокировок — не преимущество базы данных. Это — нежелательное свойство. Тот факт, что СУБД поддерживает эскалацию блокировок, означает, что ее механизм блокирования расходует слишком много ресурсов, что особенно ощутимо при управлении сотнями блокировок. В СУБД Oracle расходы ресурсов в случае одной или миллиона блокировок одинаковы, — ресурсы просто не расходуются.

Типы блокировок

Ниже перечислены пять основных классов блокировок в Oracle. Первые три — общие (используются во всех базах данных Oracle), а две остальные — только в OPS (Oracle Parallel Server — параллельный сервер).

• Блокировки DML (DML locks). Операторы SELECT, INSERT, UPDATE и DELETE. К блокировкам ЯМД относятся, например, блокировки строки данных или блокировка на уровне таблицы, затрагивающая все строки таблицы.

• Блокировки DDL (DDL locks). Операторы CREATE, ALTER и так далее. Блокировки ЯОД защищают определения структур объектов.

• Внутренние блокировки (internal locks) и защелки (latches). Это блокировки, используемые сервером Oracle для защиты своих внутренних структур данных. Например, разбирая запрос и генерируя оптимизированный план его выполнения, сервер Oracle блокирует с помощью защелки библиотечный кэш, чтобы поместить в него этот план для использования другими сеансами. Защелка — это простое низкоуровневое средство обеспечения последовательности обращений, используемое сервером Oracle, и по функциям аналогичное блокировке.

• Распределенные блокировки (distributed locks). Эти блокировки используются сервером OPS для согласования ресурсов машин, входящих в кластер. Распределенные блокировки устанавливаются экземплярами баз данных, а не отдельными транзакциями.

• Блокировки параллельного управления кэшем (PCM — Parallel Cache Management Locks). Такие блокировки защищают блоки данных в кэше при использовании их несколькими экземплярами.

DML-блокировки

Блокировки DML позволяют гарантировать, что в каждый момент времени только одному сеансу позволено изменять строку и что не может быть удалена таблица, с которой работает сеанс. Сервер Oracle автоматически, более или менее прозрачно для пользователей, устанавливает эти блокировки по ходу работы.

TX - блокировки транзакций

Блокировка TX устанавливается, когда транзакция инициирует первое изменение, и удерживается до тех пор, пока транзакция не выполнит оператор COMMIT или ROLLBACK. Она используется как механизм организации очереди для сеансов, ожидающих завершения транзакции. Каждая измененная или выбранная с помощью SELECT FOR UPDATE строка будет "указывать" на соответствующую блокировку TX. Казалось бы, это должно повлечь большие расходы ресурсов, но на самом деле этого не происходит. Чтобы понять, почему, необходимо разобраться, где "живут" блокировки и как сервер ими управляет. В Oracle блокировки хранятся как атрибут данных. У сервера Oracle нет традиционного диспетчера блокировок, поддерживающего длинный список со всеми строками, заблокированными в системе. Другие СУБД делают именно так, поскольку для них блокировки — дорогостоящий ресурс, за использованием которого надо следить. Чем больше блокировок, тем сложнее ими управлять, отсюда и заботы о том, "не слишком ли много" блокировок используется в системе.

Если бы сервер Oracle имел традиционный диспетчер блокировок, при блокировании строки нужно было бы выполнить примерно такую последовательность действий.

1. Найти адрес строки, которую необходимо заблокировать.

2. Подключиться к диспетчеру блокировок (необходимо выполнять по очереди, поскольку используются общие структуры в памяти.)

3. Заблокировать список.

4. Просмотреть список, чтобы проверить, не заблокирована ли эта строка другим сеансом.

Блокирование и одновременный доступ

5. Создать в списке новую запись, фиксирующую факт блокирования строки.

6. Разблокировать список.

Теперь, когда строка заблокирована, ее можно изменять. При последующей фиксации изменений необходимо:

7. Снова подключиться к диспетчеру.

8. Заблокировать список блокировок.

9. Найти в списке и снять все установленные блокировки.

10. Разблокировать список.

Как видите, чем больше установлено блокировок, тем больше времени потребуется для изменения данных и фиксации этих изменений. Поэтому сервер Oracle поступает примерно так:

1. Находит адрес строки, которую необходимо заблокировать.

2. Переходит на эту строку.

3. Блокирует ее (ожидая снятия блокировки, если она уже заблокирована и при этом не используется опция NOWAIT).

Поскольку блокировка хранится как атрибут данных, серверу Oracle не нужен традиционный диспетчер блокировок. Транзакция просто переходит к соответствующим данным и блокирует их (если они еще не заблокированы).

При блокировании строки данных в Oracle с блоком данных связывается идентификатор транзакции, причем остается там после снятия блокировки. Этот идентификатор уникален для нашей транзакции и задает номер сегмента отката, слот и номер изменения (sequence number). Оставляя его в блоке, содержащем измененную строку, мы какбы говорим другим сеансам, что "эти данные принадлежат нам" (не все данные в блоке,только одна строка, которую мы меняем). Когда другой сеанс обращается к блоку, он "видит" идентификатор блокировки и, "зная", что он представляет транзакцию, определяет, активна ли еще транзакция, установившая блокировку. Если транзакция уже закончена, сеанс может получить данные "в собственность". Если же транзакция активна, сеанс "попросит" систему уведомить его о завершении транзакции. Таким образом, имеется механизм организации очереди: сеанс, нуждающийся в блокировке, будет помещенв очередь в ожидании завершения транзакции, после чего получит возможность работать с данными.

Информация о блокировках и транзакциях является частью служебной информации блока. Базовый формат предусматривает в начале блока "системное" пространство для хранения таблицы транзакций для этого блока. Такая таблица транзакций включает записи для каждой "реальной" транзакции, заблокировавшей те или иные данные в этом блоке. Размер этой структуры управляется двумя атрибутами хранения, задаваемыми в операторе CREATE при создании объекта:

• INITRANS — первоначальный, заранее заданный размер этой структуры. Стандартное значение: 2 — для индексов и 1 — для таблиц.

• MAXTRANS — максимальный размер, до которого может разрастаться эта структура. Стандартное значение: 255.

Итак, каждый блок начинает стандартно свое существование с одним или двумя слотами для транзакций. Количество активных транзакций, которые могут одновременно работать с блоком, ограничивается значением MAXTRANS и доступностью пространства в блоке. Невозможно достичь 255 одновременных транзакций в блоке, если там нет места для роста данной структуры.

TM — блокировки

Такие блокировки позволяют быть уверенным, что структура таблицы не изменится при изменении ее содержимого. Рассмотрим такой пример. При изменении таблицы на нее устанавливается блокировка TM; это предотвращает применение к ней операторов DROP или ALTER другим сеансом. Если сеанс пытается применить оператор ЯОД к таблице, на которую другой сеанс установил блокировку, он получит сообщение об ошибке.

Интересная особенность блокировки TM: общее количество блокировок TM, поддерживаемых в системе, конфигурируется администратором. Его можно даже установить равным нулю. Это не означает, что база данных становится доступной только для чтения (так как блокировки не поддерживаются), — в ней не разрешены операторы DDL.

Можно также лишить возможности получать блокировки TM для отдельных объектов с помощью оператора ALTER TABLE имя_таблицы DISABLE TABLE LOCK.

DDL-блокировки

Блокировки DDL автоматически устанавливаются на объекты в ходе выполнения операторов DDL для защиты их от изменения другими сеансами. Например, при выполнении DDL-оператора ALTER TABLE T на таблицу T будет установлена исключительная блокировка DDL, что предотвращает установку блокировок DDL и TM на эту таблицу другими сеансами. Блокировки DDL удерживаются на период выполнения оператора DDL и снимаются сразу по его завершении. Это делается путем помещения операторов DDL в неявные пары операторов

фиксации (или фиксации и отката). Вот почему операторы DDL в Oracle всегда фиксируются. Операторы CREATE, ALTER и т.д. фактически выполняются, как показано в следующем псевдокоде:

```
Begin
Commit;
Оператор DDL
Commit;
Exception
When others then rollback;
End;
```

Поэтому операторы DDL всегда фиксируют транзакцию, даже если завершаются неудачно. Выполнение оператора DDL начинается с фиксации. Помните об этом. Сначала выполняется фиксация, чтобы в случае отката не пришлось откатывать предыдущую часть транзакции. При выполнении оператора DDL фиксируются все выполненные ранее изменения, даже если сам оператор DDL выполнен неудачно. Если должен быть выполнен оператор DDL, но не требуется, чтобы он зафиксировал существующую транзакцию, можно использовать автономную транзакцию.

Имеется три типа блокировок DDL:

• Исключительные блокировки DDL. Они предотвращают установку блокировок DDL или TM DML другими сеансами. Это означает, что можно запрашивать таблицу в ходе выполнения оператора DDL, но нельзя ее изменять.

• Разделяемые блокировки DDL. Они защищают структуру соответствующего объекта от изменения другими сеансами, но разрешают изменять данные.

• Нарушаемые блокировки разбора (breakable parse locks). Они позволяют объекту, например плану запроса, хранящемуся в кэше разделяемого пула, зарегистрировать свою зависимость от другого объекта. При выполнении оператора DDL, затрагивающего заблокированный таким образом объект, сервер Oracle получает список объектов, зарегистрировавших свою зависимость, и помечает их как недействительные. Вот почему эти блокировки — "нарушаемые": они не предотвращают выполнение операторов DDL.

Большинство операторов ЯОД устанавливает исключительную блокировку ЯОД. При выполнении оператора, подобного

```
Alter table t add new_column date;
```

таблица T будет недоступна для изменения, пока оператор выполняется. К таблице в этот период можно обращаться с помощью оператора SELECT, но другие действия, в том числе операторы DDL, блокируются. Некоторые операторы DDL могут выполняться без установки блокировок DDL. Например, можно выполнить:

```
create index t_idx on t(x) ONLINE;
```

Ключевое слово ONLINE изменяет метод построения индекса. Вместо установки исключительной блокировки DDL, предотвращающей изменения данных, Oracle попытается установить на таблицу низкоуровневую блокировку TM. Это предотвращает изменения структуры с помощью операторов DDL, но позволяет нормально выполнять операторы DML. Сервер Oracle достигает этого путем записи в таблице изменений, сделанных в ходе выполнения оператора DDL, и учитывает эти изменения в новом индексе, когда завершается его создание. Это существенно увеличивает доступность данных.

Другие типы операторов DDL устанавливают разделяемые блокировки DDL. Они устанавливаются на объекты, от которых зависят скомпилированные хранимые объекты, типы процедур и представлений. Например, если выполняется оператор:

```
Create view MyView as
select *
from emp, dept
where emp. deptno = dept.deptno;
```

разделяемые блокировки DDL будут устанавливаться на таблицы EMP и DEPT на все время выполнения оператора CREATE VIEW. Мы можем изменять содержимое этих таблиц, но не их структуру.

Последний тип блокировок DDL — нарушаемые блокировки разбора. Когда сеанс разбирает оператор, блокировка разбора устанавливается на каждый объект, упоминаемый в этом операторе. Эти блокировки устанавливаются, чтобы разобранный и помещенный в кэш оператор был признан недействительным (и выброшен из кэша в разделяемой памяти), если один из упоминаемых в нем объектов удален или изменена его структура.

Защелки и внутренние блокировки

Защелки и внутренние блокировки (enqueues) — простейшие средства обеспечения очередности доступа, используемые для координации многопользовательского доступа к общим структурам данных, объектам и файлам.

Защелки — это блокировки, удерживаемые в течение очень непродолжительного времени, достаточного, например, для изменения структуры данных в памяти. Они используются для защиты определенных структур памяти, например буферного кэша или библиотечного кэша в разделяемом пуле. Защелки обычно запрашиваются системой в режиме ожидания. Это означает, что, если защелку нельзя установить, запрашивающий сеанс прекращает работу ("засыпает") на короткое время, а затем пытается повторить операцию. Другие защелки могут запрашиваться в оперативном режиме, т.е. процесс будет делать что-то другое, не ожидая возможности установить защелку. Поскольку возможности установить защелку может ожидать несколько запрашивающих сеансов, одни из них будут ожидать дольше, чем другие. Защелки выделяются случайным образом по принципу "кому повезет". Сеанс, запросивший установку защелки сразу после освобождения ресурса, установит ее. Нет очереди ожидающих освобождения защелки — есть просто "толпа" пытающихся ее получить.

Для работы с защелками Oracle использует неделимые инструкции типа "проверить и установить". Поскольку инструкции для установки и снятия защелок — неделимые, операционная система гарантирует, что только один процесс сможет установить защелку. Поскольку это делается одной инструкцией, то происходит весьма быстро. Защелки удерживаются непродолжительное время, причем имеется механизм очистки на случай, если владелец защелки "скоропостижно скончается", удерживая ее. Эта очистка обычно выполняется процессом PMON.

Внутренние блокировки — более сложное средство обеспечения очередности доступа, используемое, например, при изменении строк в таблице базы данных. В отличие от защелок, они позволяют запрашивающему "встать в очередь" в ожидании освобождения ресурса. Запрашивающий защелку сразу уведомляется, возможно ли это. В случае внутренней блокировки запрашивающий блокируется до тех пор, пока не сможет эту блокировку установить. Таким образом, внутренние блокировки работают медленнее защелок, но обеспечивают гораздо большие функциональные возможности. Внутренние блокировки можно устанавливать на разных уровнях, поэтому можно иметь несколько "разделяемых" блокировок и блокировать с разными уровнями "совместности".

Блокирование вручную. Блокировки, определяемые пользователем

До сих пор мы рассматривали в основном блокировки, устанавливаемые сервером Oracle без нашего вмешательства. При изменении таблицы сервер Oracle устанавливает на нее блокировку TM, чтобы предотвратить ее удаление другими сеансами. В изменяемых блоках оставляют блокировки TX — благодаря этому другие сеансы "знают", что с данными работают. Сервер использует блокировки DDL для защиты объектов от изменений по ходу их изменения сеансом. Он использует защелки и внутренние блокировки для защиты собственной структуры. Теперь давайте посмотрим, как включиться в процесс блокирования. У нас есть следующие возможности:

• блокирование данных вручную с помощью оператора SQL;

• создание собственных блокировок с помощью пакета DBMS_LOCK.

Рассмотрим, зачем могут понадобиться эти средства.

Блокирование вручную

Мы уже описывали несколько случаев, когда может потребоваться блокирование вручную. Основным методом явного блокирования данных вручную является использование оператора SELECT...FOR UPDATE. Мы использовали его в предыдущих примерах для решения проблемы потерянного изменения, когда один сеанс может переписать изменения, сделанные другим сеансом. Мы видели, что этот оператор используется для установки очередности доступа к подчиненным записям, диктуемой бизнес-правилами.

Данные можно также заблокировать вручную с помощью оператора LOCK TABLE. На практике это используется редко в силу особенностей такой блокировки. Этот оператор блокирует таблицу, а не строки в ней. При изменении строк они "блокируются" как обычно. Так что это — не способ экономии ресурсов (как, возможно, в других реляционных СУБД). Оператор LOCK TABLE IN EXCLUSIVE MODE имеет смысл использовать при выполнении большого пакетного изменения, затрагивающего множество строк таблицы, и необходима уверенность, что никто не "заблокирует" это действие. Блокируя таким способом таблицу, можно быть уверенным, что все изменения удастся выполнить без блокирования другими транзакциями. Но приложения с оператором LOCK TABLE встречаются крайне редко.

Создание собственных блокировок

Сервер Oracle открывает для разработчиков свои механизмы блокирования для обеспечения очередности доступа с помощью пакета DBMS_LOCK.

Например, этот пакет может применяться для обеспечения последовательного доступа к ресурсу, внешнему по отношению к серверу Oracle. Пусть используется подпрограмма пакета UTL_FILE, позволяющая записывать информацию в файл в файловой системе сервера. Предположим, разработана общая подпрограмма передачи сообщений, вызываемая приложениями для записи сообщений. Поскольку файл является внешним, сервер Oracle не может координировать доступ нескольких пользователей, пытающихся его менять одновременно. В таких случаях как раз и пригодится пакет DBMS_LOCK. Прежде чем открывать файл, записывать в него и закрывать, можно устанавливать блокировку с именем, соответствующим имени файла, в исключительном режиме, а после закрытия файла вручную снимать эту блокировку. В результате только один пользователь в каждый момент времени сможет записывать сообщение в этот файл. Всем остальным придется ждать. Пакет DBMS_LOCK позволяет вручную снять блокировку, когда она уже больше не нужна, или дождаться автоматического ее снятия при фиксации транзакции, или сохранить ее до завершения сеанса.

Влияние стандартов ANSI

Если все СУБД соответствуют стандарту SQL92, они должны быть одинаковы. Так считают многие. На самом деле это во многом является мифом.

SQL92 — это стандарт ANSI/ISO для СУБД. Он является развитием стандарта ANSI/ISO SQL89. Этот стандарт задает язык (SQL) и поведение (транзакции, уровни изолированности и т.д.) для СУБД. Многие коммерческие СУБД соответствуют стандарту SQL92. Однако это немного значит для переносимости запросов и приложений.

Стандарт SQL92 имеет четыре уровня.

• Начальный. Именно этому уровню соответствует большинство предлагаемых СУБД. Этот уровень является незначительным развитием предыдущего стандарта, SQL89. Ни одна СУБД не сертифицирована по более высокому уровню. Более того, фактически Национальный институт стандартов и технологий (National Institute of Standards and Technology — NIST),

агенство, сертифицировавшее соответствие стандартам SQL, сертификацией больше не занимается.

• Переходный. С точки зрения поддерживаемых возможностей это что-то среднее между начальным и промежуточным уровнем.

• Промежуточный. Этот уровень добавляет много возможностей, в том числе (этот список далеко не исчерпывающий):

- динамический SQL;
- каскадное удаление для обеспечения целостности ссылок;
- типы данных DATE и TIME;
- домены;
- символьные строки переменной длины;
- выражения CASE;
- функции CAST для преобразования типов данных.

• Полный. Добавляет следующие возможности (этот список тоже не исчерпывающий):

- управление подключением;
- тип данных BIT для битовых строк;
- отложенная проверка ограничений целостности;
- производные таблицы в конструкции FROM;
- подзапросы в конструкции CHECK;
- временные таблицы.

В стандарт начального уровня не входят такие конструкции, как внешние соединения, новый синтаксис для внутренних соединений и т.д. Переходный уровень требует поддержки соответствующего синтаксиса внешнего и внутреннего соединения. Промежуточный уровень добавляет новые возможности, а полный и представляет собой, собственно, SQL92. В большинстве книг по SQL92 не различаются эти уровни поддержки, что сбивает с толку. В них демонстрируется, как должна работать "идеальная" СУБД, полностью реализующая стандарт SQL92. В результате нельзя взять книгу по SQL92 и применить представленные в ней приемы к СУБД, соответствующей стандарту SQL92.

Например, в СУБД SQL Server предлагаемый стандартом синтаксис "внутреннего соединения" в SQL-операторах поддерживается, а в СУБД Oracle — нет. Но обе эти СУБД соответствуют стандарту SQL92. В СУБД Oracle можно выполнять внешние и внутренние соединения, но делать это надо не так, как в SQL Server. В результате начальный уровень стандарта SQL92 мало что дает, а при использовании средств более высоких уровней возможны проблемы при переносе на другую СУБД.

Не надо бояться использовать специфические средства конкретной СУБД. В каждой СУБД есть свой набор уникальных возможностей, и в любой СУБД можно найти способ выполнить необходимое действие. Используйте в текущей СУБД лучшее и реализуйте новые компоненты при переходе на другие СУБД.

Используйте соответствующие приемы программирования, максимально изолирующие остальную часть приложения от этих изменений. Эти же приемы программирования применяются разработчиками переносимых приложений, поддерживающих несколько ОС.

Цель в том, чтобы в полной мере использовать имеющиеся средства, но при этом иметь возможность менять реализацию в каждом конкретном случае.

Например, типичная функция многих приложений баз данных — генерация уникального ключа для каждой строки. При вставке строки система должна автоматически сгенерировать ключ. В Oracle для этого предлагается объект базы данных — последовательность (SEQUENCE). В Informix имеется тип данных SERIAL. Sybase и SQL Server поддерживают тип данных IDENTITY. В каждой СУБД имеется способ решить эту задачу. Однако методы решения различны, различны и возможные последствия их применения. Поэтому знающий разработчик может выбрать один из двух вариантов:

• разработать метод генерации уникального ключа, полностью независимый от СУБД;

• согласиться с разными реализациями и использовать разные методы генерации ключей в зависимости от СУБД.

Теоретическое преимущество первого подхода состоит в том, что при переходе с одной СУБД на другую ничего менять не придется. Однако недостатки такого решения настолько велики, что делают его практически неприемлемым. Для создания полностью независимого от СУБД процесса придется создать таблицу вида:

```
create table id_table (id_name varchar(30), id_value number);
insert into id_table values ('MY_KEY', O) ;
```

Затем для получения нового ключа необходимо выполнить следующий код:

```
update id_table set id_value = id_value + 1 where id_name = 'MY_KEY';
select id_value from id_table where id_name = 'MY_KEY';
```

Выглядит он весьма просто, но выполнять подобную транзакцию в каждый момент времени может только один пользователь. Необходимо изменить соответствующую строку, чтобы увеличить значение счетчика, а это приведет к поочередному выполнению операций. Не более одного сеанса в каждый момент времени будет генерировать новое значение ключа. Проблема осложняется тем, что реальные транзакции намного больше транзакции, показанной выше. Показанные в примере операторы UPDATE и SELECT — лишь два из множества операторов, входящих в транзакцию. Необходимо еще вставить в таблицу строку с только что сгенерированным ключом и выполнить необходимые действия для завершения транзакции. Это упорядочение доступа будет огромным ограничивающим фактором для масштабирования.

Правильное решение этой проблемы состоит в использовании для каждой СУБД соответствующего кода. В Oracle (предполагается, что уникальный ключ необходимо генерировать для таблицы Т) лучшим способом будет:

```
create table t (pk number primary key, . . . ) ;
create sequence t_seq;
create trigger t_trigger before insert on t for each row
begin
select t_seq.nextval into :new.pk from dual;
end;
```

В результате каждая вставляемая строка автоматически и незаметно для приложения получит уникальный ключ. Тот же эффект можно получить и в других СУБД с помощью их типов данных — синтаксис оператора создания таблицы изменится, а результат будет тем же. Мы использовали второй вариант — специфические средства каждой СУБД для неблокируемой, высокопараллельной генерации уникального ключа, что, однако, не потребовало реальных изменений в коде приложения — все необходимые действия выполнены операторами ЯОД.

Помимо синтаксических различий в языке SQL, различаются реализации операторов, различной будет и производительность выполнения одного и того же запроса, есть проблемы управления одновременным доступом, уровней изолированности транзакций, согласованности запросов и т.д. В стандарте SQL92 попытались дать четкие определения того, как должна выполняться транзакция, как должны обеспечиваться уровни изолированности, но в конечном итоге в разных СУБД результаты получаются различными. Все это связано с реализацией. В одной СУБД приложение будет вызывать взаимные блокировки и заблокирует все, что можно. В другой СУБД это же приложение не вызывает никаких проблем и работает отлично. В одной СУБД блокирование (физически упорядочивающее обращения) намеренно использовалось в приложении, а при его переносе в другую СУБД, где блокирования нет, получается неверный ответ. Чтобы перенести готовое приложение в другую СУБД, требуется много труда и усилий, даже если при первоначальной разработке неукоснительно соблюдался стандарт.

Резюме

Давайте кратко повторим ключевые моменты. Если вы разрабатываете ПО для СУБД Oracle:

• Вы должны понимать архитектуру Oracle. Не требуется знать ее настолько, чтобы переписать сервер, но достаточно хорошо, чтобы понимать последствия использования тех или иных возможностей.

• Необходимо понимать, как выполняется блокирование и управление одновременным доступом, и учитывать, что в каждой СУБД это реализуется по-разному. Без этого понимания СУБД будет давать "неверные" ответы и у вас будут большие проблемы с конфликтами доступа и, как следствие, — низкая производительность.

• Не воспринимайте СУБД как черный ящик, устройство которого понимать не обязательно. СУБД — самая важная часть большинства приложений. Ее игнорирование приводит к фатальным последствиям.

• Не изобретайте велосипед. Максимально используйте возможностей СУБД Oracle.

• Решайте проблемы как можно проще, максимально используя встроенные возможности СУБД Oracle.

Средства Oracle по обеспечению безопасности и целостности баз данных

Приложение

Основные объекты Oracle

Oracle поддерживает реляционную модель данных, поэтому естественно, что к числу основных объектов базы данных относятся: таблица, представление и пользователь.

Пользователь (USER) — объект, обладающий возможностью создавать и использовать другие объекты Oracle, а также запрашивать выполнение функций сервера. К числу таких функций относится организация сессии, изменение состояния базы данных и др. С пользователем Oracle связана схема (SCHEMA), которая является логическим набором объектов базы данных, таких, как таблицы, последовательности, хранимые программы, принадлежащие этому пользователю. Схема имеет только одного пользователя-владельца, ответственного за создание и удаление этих объектов. При создании пользователем первого объекта неявно создается соответствующая схема. При создании им других объектов они по умолчанию становятся частью этой схемы. Для просмотра объектов схемы текущего пользователя можно использовать представление словаря данных USER_OBJECTS.

При массовом выполнении DDL-предложений можно создать несколько объектов и назначить для них привилегии за одну операцию, используя оператор CREATE SCHEMA. Оператор CREATE SCHEMA применяется тогда, когда требуется гарантировать успешное создание всех объектов и назначение привилегий за одну операцию. Если при создании объектов произошла ошибка, происходит возвращение к исходному состоянию.

Схема может содержать следующие объекты: кластеры, связи баз данных, триггеры, библиотеки внешних процедур, индексы, пакеты, последовательности, хранимые функции и процедуры, синонимы, таблицы, представления, снимки, объектные таблицы, объектные типы, объектные представления.

Объекты схемы могут состоять из других объектов, называемых подобъектами схемы. К ним относятся столбцы таблиц и представлений, секции таблиц, ограничения целостности, триггеры, пакетные процедуры и функции и другие элементы, хранимые в пакетах (курсоры, типы и т. п). К объектам, не принадлежащим схеме, но хранимым в базе данных, относятся каталоги, профили, роли, сегменты отката, табличные области и пользователи.

Таблица (TABLE) является базовой структурой реляционной модели. Как известно, вся информация в реляционной базе данных хранится в таблицах. Полное имя таблицы в базе данных состоит из имени схемы и собственно имени таблицы. Таблицы состоят из множества поименованных столбцов или атрибутов. Множество допустимых значений атрибута называют доменом значений или просто доменом. Множество допустимых значений столбца также может быть уточнено с помощью статических ограничений целостности.

Таблицы могут быть связаны между собой отношениями ссылочной целостности. Таблица может быть пустой или состоять из одной или более строк значений атрибутов. Строки значений атрибутов таблицы называются также кортежами. Для повышения скорости доступа к данным таблица может быть индексно организована (INDEX-ORGANIZED TABLE). Физическое пространство для хранения данных таблицы выделяется частями, называемыми экстентами. Размеры начального и дополнительных экстентов определяются при создании таблицы.

Представление (VIEW) — это поименованная, динамически поддерживаемая сервером выборка из одной или нескольких таблиц. По сути, представление — это производное множество строк, которое является результатом выполнения некоторого запроса к базовым таблицам. В словаре данных хранится только определение представления и, когда в операторе SQL встречается название представления, Oracle обращается к словарю за определением и подставляет его в исходный запрос. Запрос, определяющий выборку, ограничивает видимые пользователем данные. Представления позволяют упростить сложные запросы и сделать более понятными их логику. Используя представления, администратор безопасности ограничивает доступную пользователю часть базы данных только теми данными, которые реально

необходимы для выполнения его работы. Представления также можно использовать для поддержки приложений при изменении структуры таблицы. Например, при добавлении нового столбца в таблицу создать представление, его не включающее.

Синоним (SYNONYM) — это альтернативное имя или псевдоним объекта Oracle, который позволяет пользователям базы данных иметь доступ к данному объекту. Синоним может быть частным и общим. Общий (PUBLIC) синоним позволяет всем пользователям базы данных обращаться к соответствующему объекту по альтернативному имени. Характерным применением общих синонимов является сокрытие информации о схеме, в которой расположен объект. Наличие синонима позволяет обращаться к объекту по имени, которое является абсолютным в масштабе базы данных. Реальная привязка объекта к некоторой схеме при этом скрыта от пользователя или приложения.

Для управления эффективностью доступа к данным Oracle поддерживает следующие объекты: индекс, табличная область, кластер и хэш-кластер.

Индекс (INDEX) — это объект базы данных, предназначенный для повышения производительности выборки данных. Индекс создается для столбцов таблицы и обеспечивает более быстрый доступ к данным за счет хранения указателей (ROWID) на месторасположение строк. При обращении к индексированному столбцу сервер по предъявляемому значению находит в индексе указатели на эти строки, а потом непосредственно обращается к ним. Если все требуемые значения столбцов имеются в индексе, обращение к таблице не происходит вовсе. Имеется несколько типов индексов — B*Tree (двоичное дерево, каждый узел которого содержит указатель на следующий и предыдущий), масочный индекс, индекс с реверсированным ключом, кластерный индекс. Подробнее эта тема будет разобрана далее.

Кластер (CLUSTER) — объект, задающий способ хранения данных нескольких таблиц, содержащих информацию, обычно обрабатываемую совместно. Строки таких таблиц, имеющие одинаковое значение в кластеризованных столбцах, хранятся в базе данных специальным образом. Кластеризация столбцов таблиц позволяет уменьшить время выполнения выборки.

При использовании **хэшированных кластеров (HASH CLUSTER)** организация таблиц базируется на результатах хэширования их первичных ключей. Для получения данных из такого кластера запрашиваемое значение ключа обрабатывается хэш-функцией, полученное значение определяет, в каком блоке кластера хранятся данные.

Табличная область (TABLESPACE) — именованная часть базы данных, используемая для распределения памяти для таблиц, индексов и других объектов. В табличную область входит один или несколько файлов. Это предоставляет возможность гибко настроить хранение данных в зависимости от порядка и интенсивности их использования. Например, можно отвести одну табличную область для таблиц, а другую — для индексов. В каждой базе данных есть табличная область SYSTEM, с которой связаны все системные объекты, например таблицы словаря данных. Доступность табличных областей может устанавливаться переводом в автономный или оперативный режим.

Для эффективного управления разграничением доступа к данным Oracle поддерживает объект роль.

Роль (ROLE) — именованная совокупность привилегий, которые могут быть предоставлены пользователям или другим ролям. Oracle поддерживает несколько предопределенных ролей. Для систем, в которых количество пользователей и приложений велико, роли могут заметно облегчить разграничение доступа, например, возможно динамически назначать роли для изменения набора привилегий пользователя при работе с различными приложениями. Также роли можно защищать паролем.

Специфичными для распределенных систем являются объекты Oracle: снимок и связь базы данных.

Снимок (SNAPSHOT) — локальная копия таблицы удаленной базы данных, которая, используется либо для тиражирования (копирования) всей или части таблицы, либо для тиражирования результата запроса данных из нескольких таблиц. Снимки могут быть модифицируемыми или предназначенными только для чтения. Снимки только для чтения

возможно периодически обновлять, отражая изменения основной таблицы. Изменения, сделанные в модифицируемом снимке, распространяются на основную таблицу и другие копии.

Связь базы данных (DATABASE LINK) — это объект базы данных, который позволяет обратиться к объектам удаленной базы данных. Имя связи базы данных можно рассматривать как ссылку на параметры механизма доступа к удаленной базе данных (имя узла, протокол и т. п.).

Сегмент отката (ROLBACK SEGMENT) — объект базы данных, предназначенный для обеспечения многопользовательской работы. В сегментах отката находятся обновляемые и удаляемые данные в пределах одной транзакции. При отмене изменений старая версия данных всегда доступна, так как находится в сегментах отката. В начале транзакции и в каждой контрольной точке текущие значения данных копируются в сегмент отката. Кроме того, сегменты отката используются при других операциях сервера. Размер и доступность сегментов отката в сильной степени влияют на производительность сервера баз данных и их настройка должна быть выполнена самым тщательным образом.

Для программирования алгоритмов обработки данных, реализации механизмов динамической поддержки целостности базы данных Oracle использует следующие объекты: процедуры, функции, пакеты, тела пакетов и триггеры.

Процедура (PROCEDURE) — это поименованный, структурированный набор конструкций языка PL/SQL, предназначенный для решения конкретной задачи.

Функция (FUNCTION) — это поименованный, структурированный набор конструкций языка PL/SQL, предназначенный для решения конкретной задачи и возвращающий значение.

Пакет (PACKAGE) — это поименованный, структурированный набор переменных, процедур, функций и других объектов, связанных функциональным замыслом. Пакет состоит из двух самостоятельных частей: заголовка и тела. Заголовок содержит описание переменных, констант, типов, процедур, функций и других конструкций языка PL/SQL. Тело пакета содержит реализацию алгоритмов процедур и функций и хранится отдельно. Например, Oracle предоставляет стандартный пакет UTL_FILE, который содержит процедуры и функции, предназначенные для организации файлового ввода-вывода из программ на языке PL/SQL.

Триггер (TRIGGER) — это хранимая процедура, которая автоматически выполняется тогда, когда происходит связанное с триггером событие. Обычно события связаны с выполнением операторов вставки, модификации и удаления данных. С помощью триггеров можно реализовать правила динамической проверки целостности данных и дополнительного контроля доступа.

Библиотека (LIBRARY) — объект базы данных, предназначенный для взаимодействия программ PL/SQL с модулями, написанными на других языках программирования.

Каталог (DIRECTORY) — объект, предназначенный для организации файлового ввода-вывода и работы с большими двоичными объектами.

Профиль (PROFILE) — объект, ограничивающий использование пользователем системных ресурсов, например процессорного времени или числа операций ввода-вывода.

База данных — это некоторый набор перманентных (постоянно хранимых) данных, используемых прикладными программными системами какого-либо предприятия.

Модель данных — это абстрактное, самодостаточное, логическое определение объектов, операторов и прочих элементов, в совокупности составляющих абстрактную машину доступа к данным, с которой взаимодействует пользователь. Упомянутые объекты позволяют моделировать структуру данных, а операторы — поведение данных.

Реализация (implementation) заданной модели данных — это физическое воплощение на реальной машине компонентов абстрактной машины, которые в совокупности составляют эту модель.

Модель данных (во втором смысле) представляет собой модель перманентных данных некоторого конкретного предприятия.

Транзакция (transaction) — это логическая единица работы (точнее, логическая единица работы базы данных), обычно включающая несколько операций базы данных

Средства манипулирования данными языка SQL

В языке SQL предусмотрено четыре ключевых слова для операций манипулирования данными: SELECT, INSERT, UPDATE и DELETE.

Для описания синтаксиса будем использовать следующие конструкции:

| - любой предшествующий | символ может быть произвольно заменен на любой следующий за | (высказывание ИЛИ);

{} – все, что включено в фигурные скобки, обрабатывается единым блоком для выбора из вариантов, разделенных символом |, или заполнения конкретными значениями;

[] – все, что заключенов квадратные скобки, является необязательными элементами;

… - конкретные значения, предшествующие символу, могут повторяться любое число раз.

Оператор SELECT

Оператор SELECT используется для выборки атрибутов одной или нескольких таблиц в соответствии с указанным критерием отбора.

Этот оператор поддерживает 3 основные операции реляционной модели: сокращение, проекция и соединение.

Описание полного синтаксиса оператора выборки достаточно велико. Поэтому будем рассматривать сокращенное множество конструкций оператора SELECT. Полное описание синтаксиса доступно в соответствующей части документации Oracle SQL Reference.

SELECT [DISTINCT | ALL] { * | { [имя_схемы.]
{имя_таблицы | имя_представления | имя_снимка }.*|
выражение [[AS] альтернативное_имя_столбца] }
[, { [имя_схемы.] { имя_таблицы | имя_представления
| имя_ снимка }.*| выражение
[[AS] альтернативное_имя_столбца] }] ... }
FROM {[имя_схемы .]{{ имя_таблицы |
имя_представления | имя_снимка } [@имя_связиБД]
} | (имя__подзапроса) }
[локальное_альтернативное_имя]
[, { [имя_схемы.] { { имя_таблицы |
имя_представления | имя_снимка } [@имя_связиБД]
}| (имя_ подзапроса) }
[локальное_альтернативное_имя]] ...
[WHERE условие]
{ { [GROUP BY выражение [, выражение] ...
[HAVING условие]] }
|{[START WITH условие] CONNECT BY условие}} ...
[{UNION | UNION ALL | INTERSECT | MINUS}
предложение_SELECT]
[ORDER BY { выражение | положение |
альтернативное_имя_столбца }
[ASC | DESC]
[, { выражение | положение |
альтернативное_имя_столбца } [ASC | DESC]] ...]
[FOR UPDATE [OF [[имя_схемы .] { имя_таблицы |
имя_представления }] имя_столбца [, [имя_схемы .]
{имя_таблицы | имя_представления }] имя_столбца
...] [NOWAIT]]

Фрагменты предложения SELECT должны быть записаны в указанном порядке.

Результатом выполнения операции выборки является мультимножество, то есть, если это не указано явно, устранение повторяющихся строк не производится. Для устранения повторяющихся строк используется ключевое слово DISTINCT. Для устранения дубликатов Oracle неявно производит сортировку и в этом случае время выполнения запроса может оказаться неожиданно большим. Поэтому не рекомендуется использовать эту конструкцию в приложениях, требующих малого времени реакции. Указание ключевого слова ALL приводит к предъявлению всех отобранных данных, включая повторяющиеся строки. По умолчанию используется значение ALL.

После ключевого слова SELECT следует список, определяющий перечень выводимых столбцов. Наличие параметра * (звездочка) означает выбор всех столбцов из всех таблиц, представлений и снимков, указанных в перечне значений ключевого слова FROM.

Конкретный выбор значения параметра {имя_таблцы | имя представления | имя_снимка}.* означает выбор всех столбцов из таблицы, представления или снимка.

Необязательный параметр имя_схемы используется для уточнения имени схемы, в которой находится соответствующий объект. По умолчанию используется схема пользователя, выполняющего запрос.

Параметр выражение заменяется на вычисляемое выражение, которое обычно базируется на данных столбцов из таблиц, представлений и снимков, указанных в перечне значений ключевого слова FROM.

Параметр альтернативное_имя_столбца задает альтернативное имя столбца или выражения для формирования заголовка при выводе ответа на запрос. Заданное значение параметра может также использоваться в выражении ORDER BY.

Ключевое слово FROM определяет таблицы, представления или снимки, из которых будут отбираться данные.

Параметр имя_связиБД устанавливает имя связи с удаленной базой данных. Если имя связи с удаленной базой данных не указано, предполагается, что соответствующий объект расположен в основной базе данных.

Параметр имя_подзапроса задает подзапрос, который в данном контексте рассматривается так же, как представление.

Параметр локальное_альтернативное_имя задает альтернативное, обычно короткое, имя, которое в контексте данного запроса является обязательным для ссылок именем соответствующей таблицы, представления или снимка.

Ключевое слово WHERE определяет логическое условие отбора данных. Если ключевое слово WHERE опущено, то возвращается декартово произведение всех таблиц, представлений и снимков, указанных в перечне значений ключевого слова FROM.

Базовые средства определения критерия отбора

Обратите внимание, что стандарт SQL допускает наличие в базе данных неопределенных значений, поэтому вычисление условия отбора должно производиться не в булевой, а в трехзначной логике со значениями TRUE, FALSE и UNKNOWN (неизвестно). Для любого предиката известно, в каких ситуациях он может порождать значение UNKNOWN.

Булевские операции AND, OR и NOT работают в трехзначной логике следующим образом:

Операция	Результат
True and Unknown	Unknown
False and Unknown	False
Unknown and Unknown	Unknown
True or Unknown	True
False or Unknown	Unknown
Unknown or Unknown	Unknown
Not Unknown	Unknown

Среди предикатов условия отбора в соответствии со стандартом SQL используются следующие предикаты: предикат сравнения, предикат BETWEEN, предикат IN, предикат LIKE, предикат IS NULL, предикат EXISTS и предикат с квантором (ALL, ANY, SOME).

Язык SQL допускает связывание предикатов логическими связками AND, OR, и NOT. Применение логического отрицания NOT возможно как к отдельному (в том числе и одноместному) предикату, так и выражению, образованному из предикатов с помощью логических связок. Для устранения неоднозначности и потенциальных ошибок при конструировании сложных критериев рекомендуется использовать круглые скобки для группирования.

Следует иметь в виду, что последовательность вычисления истинности предикатов умышленно не определена и меняется в зависимости от выражения. Как только значение выражения вычислено на основе некоторого набора предикатов, истинность остальных предикатов не вычисляется. В зависимости от плана выполнения запроса последовательность вычислений также изменяется.

Для организации группирования отобранных данных с целью их совместной обработки используется ключевое слово GROUP BY. Совместная обработка данных обычно сводится к вычислению некоторой функции: суммы, среднего значения, числа элементов множества отобранных значений и т. п.

Использование ключевого слова GROUP BY приводит к тому, что оператор SELECT выдает одну производную строку для каждой группы строк, формируемых на основе одинаковых значений для столбцов или выражений.

Синтаксис конструкции группирования строк:

GROUP BY выражение [, выражение]

[HAVING условие]

Элемент выражение может быть атрибутом, константой или функцией от них.

Ключевое слово HAVING используется для уточнения, какие группы из GROUP BY будут включаться в окончательный результат.

Групповые функции возвращают результаты, вычисленные по группе строк, которые сформированы запросом с предложением GROUP BY. Рассмотрим некоторые из них.

Функция вычисления среднего значения AVG возвращает среднее значение числового аргумента выражение, не включая в вычисления значения NULL. Функция использует следующий синтаксис:

AVG([DISTINCT | ALL] выражение)

Функция вычисления суммы SUM возвращает сумму значений числовых атрибутов, не включая в вычисления значения NULL. Функция использует следующий синтаксис:

SUM([DISTINCT | ALL] выражение)

Функция подсчета числа отобранных строк COUNT возвращает количество выбранных строк. Особый вариант использования функции COUNT(*) возвращает число строк в таблице, включая дубликаты и атрибуты с неопределенными значениями. Функция использует следующий синтаксис:

COUNT([DISTINCT | ALL] выражение | *)

Ключевые слова START WITH и CONNECT BY задают иерархический порядок отбора данных запроса. Конкретная иерархическая упорядоченность задается параметрами условие.

Ключевые слова UNION, UNION ALL, INTERSECT, MINUS задают теоретико-множественные операции объединения результатов нескольких запросов, сформированных в соответствии со значением параметра предложением SELECT. Для объединенных результатов не допускается использование конструкции FOR UPDATE.

UNION — формальное объединение результатов всех исходных запросов в виде отношения (то есть с устранением повторяющихся строк);

UNION ALL — формальное объединение результатов всех исходных запросов с сохранением повторяющихся строк;

INTERSECT — формальное пересечение; включает строки, входящие во все результаты составляющих запросов, повторяющиеся строки исключаются;

MINUS — результирующее множество содержит все строки, вошедшие в результат первого запроса, но не вошедшие в результат второго; повторяющиеся строки исключаются.

Для сортировки результатов запроса по возрастанию или убыванию используется ключевое слово ORDER BY.

ORDER BY {выражение|положение|альтернативное_имя_столбца}
[ASC|DESC]

Параметр выражение определяет значение, по которому выполняется сортировка. Базис сортировки может также быть указан параметром положение, то есть порядковым номером в списке вывода (задаваемом после ключевого слова SELECT). По умолчанию используется сортировка по возрастанию (ASC).

Конструкция FOR UPDATE определяет необходимость блокировки отобранных строк. Необязательное ключевое слово OF уточняет перечень таблиц или представлений, данные из которых должны быть заблокированы.

Необязательное ключевое слово NOWAIT указывает на то, что в случае, если требуемые для блокировки строки недоступны (то есть заблокированы другим процессом), возвращается сообщение об ошибке (как правило, ORA-00054). Если ключевое слово NOWAIT не указано, то выполнение запроса будет приостановлено.до тех пор, пока не будут освобождены все требуемые для блокировки строки.

Операция вставки строк

Операция INSERT используется для вставки строк в таблицу или базовые таблицы представления. Предложение

INSERT имеет следующий синтаксис:

INSERT INTO {[имя_схемы.]имя_таблицы |
имя_представления } [@имя_связиБД] |
(подзапрос_1) }
[(имя_столбца [, имя_столбца] ...)]
{VALUES (выражение [, выражение] ...) |
подзапрос_2}

При вставке строк с использованием представления строки добавляются в базовые таблицы представления. Необязательный параметр имя_схемы используется для уточнения имени схемы, в которой находится соответствующий объект Oracle. По умолчанию используется схема пользователя, выполняющего операцию.

Параметр имя_связиБД устанавливает имя связи с удаленной базой данных. Если имя связи с удаленной базой данных не указано, предполагается, что соответствующий объект Oracle расположен в основной базе данных.

Параметр подзопрос_1 задает подзапрос, который в данном контексте рассматривается как представление.

Параметр выражение заменяется на вычисляемое выражение, обычно базирующееся на данных столбцов из таблиц, представлений и снимков, указанных в перечне значений ключевого слова INTO.

Параметр подзапрос_2 задает подзапрос, который формирует множество вводимых строк.

Если значение какого-либо атрибута не определено и описание таблицы не запрещает наличие неопределенного значения данного атрибута, то может использоваться вставка строки с неопределенным значением атрибута.

CREATE TABLE Item

(ID Integer,
Name Varchar2(50));

INSERT INTO Item VALUES(1, NULL);
1 row created.

Операция удаления строк

Операция DELETE используется для удаления строк из таблицы или базовых таблиц представления. Предложение DELETE имеет следующий синтаксис:

DELETE [FROM]
{ [имя_схемы.] { имя_таблицы | имя_представления } | подзапрос }
[альтернативное_имя] [WHERE условие]

При удалении строк с указанием представления удаляются строки из базовых таблиц представления. Необязательный параметр имя_схемы используется для уточнения имени схемы, в которой находится соответствующий объект Oracle. По умолчанию используется схема пользователя, выполняющего операцию.

Параметр подзапрос задает подзапрос, который в данном контексте рассматривается как представление.

Если ключевое слово WHERE отсутствует, то из таблицы удаляются все строки. Если же ключевое слово WHERE используется, удаляются строки, для которых условие выполняется. Все удаляемые строки и соответствующие индексы освобождают занимаемую ими память.

DELETE FROM Item WHERE ID < = 100
2 rows deleted.
DELETE FROM Item;
0 row(s) deleted.

Операция модификации строк

Операция осуществляет модификацию строк из таблицы, базовой таблицы представления или снимка. Предложение UPDATE имеет следующий синтаксис:

UPDATE {[имя_схемы.]{ имя_таблицы |
имя_представления | имя_снимка }
[@имя_связиБД] | подзапрос_1 }
[альтернативное_имя]
SET { [(имя_столбца [, имя_столбца] ...) =
(подзапрос_2)] |
имя_столбца = { выражение | подзапрос_3 } } ...
[WHERE условие]

При модификации строк с параметром имя_представления изменяются строки из базовых таблиц представления. Необязательный параметр имя_схемы используется для уточнения имени схемы, в которой находится соответствующий объект Oracle. По умолчанию используется схема пользователя, выполняющего операцию.

Параметр имя_связиБД устанавливает имя связи с удаленной базой данных. Если имя связи с удаленной базой данных не указано, предполагается, что соответствующий объект Oracle расположен в локальной базе данных.

Параметры подзапрос_1, подзапрос_2, подзапрос_3 задают подзапросы, которые в данном контексте рассматривается как представление.

Параметр выражение представляет собой вычисляемое выражение, обычно базирующееся на данных столбцов из таблиц.

Рассмотрим несколько примеров, иллюстрирующих применение оператора UPDATE.

UPDATE Item SET Name = 'Trumpet' WHERE ID = 5;

UPDATE Item SET ID = ID+1000

WHERE Name LIKE 'A%';

(Предикат LIKE применим только к полям типа CHAR, VARCHAR и VARCHAR2. Предикат принимает истинное значение при вхождении определенной подстроки в строку. В качестве механизма формирования условия используется шаблон, состоящий из специальных символов и обычных символов используемой кодировки. В роли специальных символов выступают:

символ подчеркивания (_), замещающий любой одиночный символ;

символ процента (%), замещающий последовательность любого числа символов (включая пустой символ)).

Указанный выше запрос приведет к увеличению всех ID товаров, начинающихся с A, на 1000.

Список литературы

1. Том Кайт. Oracle для профессионалов. DiaSoft, 2003.
2. Скот Урман. Oracle 9i. Программирование на языке PL/SQL. Лори (Oracle Press), 2004.
3. Лилиан Хоббс, Сьюзан Хилсон, Шилпа Лоуленд. Oracle 9iR2. Разработка и эксплуатация хранилищ баз данных. Кудиц-Образ, 2004.
4. С.Н. Смирнов, И.С. Задворьев. Работаем с Oracle. Гелиос АРВ, 2002.
5. Официальная документация Oracle (http://www.oracle.com).
6. Кристофер Аллен. Oracle PL/SQL 101. Лори (Oracle Press), 2001.
7. Дейв Энсор, Йен Стивенсон. Oracle. Проектирование баз данных. BHV, 1999.
8. Рик Гринвальд, Роберт Стаковьян, Гэри Додж, Дэвид Кляйн, Бен Шапиро, Кристофер Дж. Челья. Программирование баз данных Oracle для профессионалов. Диалектика, 2007.
9. Джеймс Пери, Джеральд Пост. Введение в Oracle 10g. Вильямс, 2006.
10. К. Дж. Дейт. Введение в системы баз данных. Вильямс, 2005.
11. С.Д. Кузнецов. Основы баз данных. Учебное пособие. Интернет-Университет Информационных Технологий. Москва, 2007.
12. В.С. Рублев Проектирование реляционной базы данных и интерфейса. Методические указания по лабораторному практикуму. Ярославль: ЯрГУ им. П.Г. Демидова, 2007.
13. intuit.ru (А.С. Грошев «Основы работы с базами данных» и др.)