Процессы. Блокирование и совместный доступ

Разбираясь с использованием памяти сервером **Oracle**, рассмотрели половину экземпляра. Оставшийся компонент архитектуры — набор процессов, образующий вторую половину экземпляра. Некоторые из этих процессов, например, процесс записи блоков в базу данных (**DBWn**) и процесс записи журнала (**LGWR**), уже упоминались. Здесь мы более детально рассмотрим функцию каждого процесса: что и почему они делают. В этом разделе "процесс" будет использоваться как синоним "потока" в операционных системах, где сервер **Oracle** реализован с помощью потоков. Так, например, если описывается процесс **DBWn**, в среде **Windows** ему соответствует поток **DBWn**.

В экземпляре **Oracle** есть три класса процессов.

- **Серверные процессы**. Процессы, которые 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 оперативного журнала повторного выполнения, удаление прекративших работу процессов и т.д.
- **Подчиненные процессы**. Они подобны фоновым процессам, но выполняют, кроме того, действия от имени фонового или серверного процесса.

Мы рассмотрим все эти процессы и постараемся выяснить, какую роль они играют в экземпляре.

Серверные процессы

Здесь мы опишем два вида серверных процессов и детально рассмотрим их архитектуру.

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 кеше или считывая данные в буферный кеш с диска. Такие серверные процессы можно назвать "рабочими лошадками" СУБД. Часто именно они потребляют основную часть процессорного времени в системе, поскольку выполняют сортировку, суммирование, соединения — в общем, почти все.

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.

С клиентским приложением скомпонованы библиотеки **Oracle**. Они обеспечивают функциональный 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 (даже в **n-звенной** архитектуре есть место для клиент-серверного взаимодействия). Сервер **Oracle** использует такую архитектуру, даже если протокол **Net8** не нужен. То есть, когда клиент и сервер работают на одной и той же машине, используется эта двухпроцессная (известная также как двухзадачная — **two-task**) архитектура. Эта архитектура обеспечивает два преимущества.

- **Удаленное выполнение**. Клиентское приложение, естественно, может работать не на той машине, где работает СУБД.
- **Изолирование адресных пространств**. Серверный процесс имеет доступ для чтения и записи к области **SGA**. Ошибочный указатель в клиентском процессе может повредить структуры данных в области **SGA**, если клиентский и серверный процессы физически взаимосвязаны.

Теперь давайте более детально рассмотрим другой тип серверных процессов — разделяемый серверный процесс. Для подключения к серверному процессу этого типа обязательно используется протокол **Net8**, даже если клиент и сервер работают на одной машине, — нельзя использовать режим **MTS** без процесса прослушивания **Net8**. Как уже описывалось ранее в этом разделе, 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.

Клиентские приложения со скомпонованными в них библиотеками **Oracle** физически подключаются к диспетчеру **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. 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.

Прежде чем перейти к остальным процессам, нужно понимать, почему поддерживается два режима подключения и когда лучше использовать каждый из них. 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. Настройка и конфигурирование режима **MTS**, хотя и несложный, но дополнительный шаг. Основное различие между этими режимами, однако, не в настройке. Оно связано с особенностями работы. При использовании выделенного сервера имеется соответствие один к одному между клиентским сеансом и серверным процессом. 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. Как было показано в главе 1, в примере с компонентами **EJB**, запускавшими продолжительную хранимую процедуру, монополизация этого ресурса может приводить как бы к зависанию системы. На представленной выше схеме имеется два разделяемых сервера. При наличии трех клиентов, более-менее одновременно пытающихся запустить 45-секундный процесс, два из них получат результат через 45 секунд, а третий — через 90 секунд. Правило номер один для режима **MTS**: убедитесь, что транзакции выполняются быстро. Они могут выполняться часто, но должны быть короткими (что обычно и бывает в системах **OLTP**). 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".

Поэтому режим **MTS** очень хорошо подходит для систем класса **OLTP**, характеризующихся короткими, но частыми транзакциями. В системе класса **OLTP** транзакции выполняются за миллисекунды, — ни одно действие не требует для выполнения более чем доли секунды. Режим **MTS** не подходит, однако, для хранилища данных. В такой системе выполняются запросы продолжительностью одна, две, пять и более минут. Для режима **MTS** это "смертельно". В системе, где 90 процентов задач относятся к классу **OLTP**, а 10 процентов — "не совсем **OLTP**", можно поддерживать одновременно выделенные и разделяемые серверы в одном экземпляре. В этом случае существенно сокращается количество процессов для пользователей **OLTP**, а "не совсем **OLTP**"-задачи не монополизируют надолго разделяемые серверы.

Если система не перегружена и нет необходимости использовать режим **MTS** для обеспечения необходимой функциональности, лучше использовать выделенный сервер. Выделенный сервер проще устанавливать, и упрощается настройка производительности. Есть ряд операций, которые можно выполнять только при подключении в режиме выделенного сервера, так что в любом экземпляре надо поддерживать либо оба режима, либо только режим выделенного сервера.

С другой стороны, если необходимо поддерживать большое количество пользователей и известно, что эксплуатировать систему придется в режиме **MTS**, я рекомендую разрабатывать и тестировать ее тоже в режиме **MTS**. Если система разрабатывалась в режиме разделяемого сервера и никогда не тестировалась в режиме **MTS**, вероятность неудачи повышается. Испытывайте систему в рабочих условиях; тестируйте ее производительность; проверьте, хорошо ли она работает в режиме **MTS**. То есть, проверьте, не монополизирует ли она надолго разделяемые серверы. Обнаруженные на стадии разработки недостатки устранить гораздо проще, чем при внедрении. Для сокращения времени работы процесса можно использовать средства расширенной обработки очередей (**Advanced Queues — AQ**), но это надо учесть в проекте приложения. Такие вещи лучше делать на этапе разработки.

Фоновые процессы

Экземпляр **Oracle** состоит из двух частей: области **SGA** и набора фоновых процессов. Фоновые процессы выполняют рутинные задачи сопровождения, обеспечивающие работу СУБД. Есть, например, процесс, автоматически поддерживающий **буферный кеш** и при необходимости записывающий блоки данных на диск. Есть процесс, копирующий заполненный файл оперативного журнала повторного выполнения в архив. Еще один процесс отвечает за очистку всех структур, которые использовались завершившимися

процессами, и т.д. 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 (как только что описанные) и решающие множество различных задач. Например, есть фоновый процесс, обеспечивающий работу внутренних очередей заданий в **Oracle**. Этот процесс контролирует очередь заданий и выполняет находящиеся в ней задания. Во многом он похож на выделенный сервер, но без подключения к клиенту. Сейчас мы рассмотрим все эти фоновые процессы, начиная с тех, которые выполняют конкретную задачу, а затем перейдем к процессам "общего назначения".

Фоновые процессы, предназначенные для решения конкретных задач:

PMON — монитор процессов

Этот процесс отвечает за очистку после нештатного прекращения подключений. Например, если выделенный сервер "падает" или, получив сигнал, прекращает работу, именно процесс **PMON** освобождает ресурсы. Процесс **PMON** откатит незафиксированные изменения, снимет блокировки и освободит ресурсы в области **SGA**, выделенные прекратившему работу процессу.

Помимо очистки после прерванных подключений, процесс **PMON** контролирует другие фоновые процессы сервера Oracle и перезапускает их при необходимости (если это возможно). Если разделяемый сервер или диспетчер сбоит (прекращает работу), процесс **PMON** запускает новый процесс (после очистки структур сбойного процесса). Процесс **PMON** следит за всеми процессами **Oracle** и либо перезапускает их, либо прекращает работу экземпляра, в зависимости от ситуации. Например, в случае сбоя процесса записи журнала повторного выполнения (**LGWR**) экземпляр надо перезапускать. Это серьезная ошибка и самое безопасное — немедленно прекратить работу экземпляра, предоставив исправление данных штатному процессу восстановления. Это происходит очень редко, и о случившемся надо немедленно сообщить службе поддержки **Oracle**.

Еще одна функция процесса **PMON** в экземпляре (версия **Oracle 8i**) — регистрировать экземпляр в процессе прослушивания протокола Net8. При запуске экземпляра процесс **PMON** опрашивает известный порт (если явно не указан другой), чтобы убедиться, запущен и работает ли процесс прослушивания. Известный/стандартный порт, используемый сервером **Oracle**, — порт 1521. А что произойдет, если процесс прослушивания запущен на другом порту? В этом случае используется тот же механизм, но адрес процесса прослушивания необходимо указать явно с помощью параметра инициализации **LOCAL_LISTENER**. Если процесс прослушивания запущен, процесс **PMON** связывается с ним и передает соответствующие параметры, например, имя службы.

SMON — монитор системы

SMON — это процесс, занимающийся всем тем, от чего "отказываются" остальные процессы. Это своего рода "сборщик мусора" для базы данных. Вот некоторые из решаемых им задач.

- **Очистка временного пространства**. С появлением по-настоящему временных табличных пространств эта задача упростилась, но она не снята с повестки дня полностью. Например, при построении индекса выделяемые ему в ходе создания экстенты помечаются как временные (**TEMPORARY**). Если выполнение оператора **CREATE INDEX** прекращено досрочно по какой-либо причине, процесс **SMON** должен эти экстенты освободить. Есть и другие операции, создающие временные экстенты, за очистку которых также отвечает процесс **SMON**.

- **Восстановление после сбоев**. Процесс **SMON** после сбоя восстанавливает экземпляр при перезапуске.

- **Дефрагментация свободного пространства**. При использовании табличных пространств, управляемых по словарю, процесс **SMON** заменяет расположенные подряд свободные экстенты одним "большим" свободным экстентом. Это происходит только в табличном пространстве, управляемом по словарю и имеющем стандартную конструкцию хранения с ненулевым значением параметра **PCTINCREASE**.

- **Восстановление транзакций, затрагивающих недоступные файлы**. Эта задача аналогична той, которая возникает при запуске базы данных. Процесс **SMON** восстанавливает сбойные транзакции,

пропущенные при восстановлении экземпляра после сбоя по причине недоступности файлов для восстановления. Например, файл мог быть на недоступном или на не смонтированном диске. Когда файл будет доступен, процесс **SMON** восстановит его.

- **Восстановление сбойного экземпляра в OPS**. В конфигурации **Oracle Parallel Server**, если одна из машин кластера останавливается (на ней происходит сбой), другая машина в экземпляре откроет файлы журнала повторного выполнения этой сбойной машины и восстановит все данные этой машины.

- **Очистка таблицы OBJ$**. **OBJ$** — низкоуровневая таблица словаря данных, содержащая записи практически для каждого объекта (таблицы, индекса, триггера, представления и т.д.) базы данных. Часто там встречаются записи, представляющие удаленные или "отсутствующие" объекты, используемые механизмом поддержки зависимостей **Oracle**. Процесс **SMON** удаляет эти ненужные строки.

- **Сжатие сегментов отката**. Процесс **SMON** автоматически сжимает сегмент отката до заданного размера.

- **"Отключение" сегментов отката**. Администратор базы данных может "отключить" или сделать недоступным сегмент отката с активными транзакциями. Активные транзакции могут продолжать использование такого отключенного сегмента отката. В этом случае сегмент отката фактически не отключается: он помечается для "отложенного отключения". Процесс **SMON** периодически пытается "действительно" отключить его, пока это не получится.

Этот список дает представление о том, что делает процесс **SMON**. Как видно из представленной выше информации о процессах, полученной с помощью команды **ps**, процесс **SMON** может со временем потребовать существенных вычислительных ресурсов (команда **ps** выполнялась на машине, где экземпляр проработал около месяца). Процесс **SMON** периодически "пробуждается" (или его "будят" другие фоновые процессы) для выполнения задач сопровождения.

RECO — восстановление распределенной базы данных

Процесс **RECO** имеет очень конкретную задачу: он восстанавливает транзакции, оставшиеся в готовом состоянии из-за сбоя или потери связи в ходе двухэтапной фиксации (**2PC**). **2PC** — это распределенный протокол, позволяющий неделимо фиксировать изменения в нескольких удаленных базах данных. Он пытается максимально снизить вероятность распределенного сбоя перед фиксацией. При использовании протокола **2PC** между **N** базами данных одна из баз данных обычно (но не всегда) та, к которой первоначально подключился клиент, становится координатором. Соответствующий сервер опрашивает остальные **N -1** серверов, готовы ли они фиксировать транзакцию. Фактически, этот сервер связывается с остальными **N - 1** серверами и просит их подготовиться к фиксации. Каждый из **N -1** серверов сообщает о своем состоянии готовности как да (**YES**) или нет (**NO**). Если любой из серверов вернул **NO**, вся транзакция откатывается. Если все серверы вернули **YES**, координатор рассылает всем **N - 1** серверам сообщение о постоянной фиксации.

Если серверы ответили **YES** и подготовились к фиксации, но до получения директивы о фактической фиксации от координатора происходит сбой сети или возникает какая-то другая ошибка, транзакция становится сомнительной (**in-doubt**) распределенной транзакцией. Протокол **2PC** старается сократить до минимума время, в течение которого это может произойти, но не может полностью предотвратить сомнительные транзакции. Если сбой произойдет в определенном месте и в определенное время, дальнейшую обработку сомнительной транзакции выполняет процесс **RECO**. Он пытается связаться с координатором транзакции, чтобы узнать ее исход. До этого транзакция остается незафиксированной. Связавшись с координатором транзакции, процесс **RECO** восстановит либо откатит ее.

Если связаться с координатором долго не удается и имеется ряд сомнительных транзакций, их можно зафиксировать или откатить вручную. Это приходится делать, поскольку сомнительная распределенная транзакция может вызвать блокирование читающих пишущими (единственный случай в СУБД Oracle). Ваш администратор базы данных должен связаться с администратором другой базы данных и попросить его определить состояние сомнительных транзакций. Затем администратор базы данных может зафиксировать или откатить их, предоставив все остальное процессу **RECO**.

CKPT — обработка контрольной точки

Процесс обработки контрольной точки вовсе не обрабатывает ее, как можно предположить по названию, — это делает процесс **DBWn**. Процесс **CKPT** просто содействует обработке контрольной точки, обновляя заголовки файлов данных. Раньше процесс **CKPT** был необязательным, но, начиная с версии 8.0, он

запускается всегда, так что он представлен в результатах выполнения команды **ps** в ОС **UNIX**. Ранее заголовки файлов данных обновлялись в соответствии с информацией о контрольной точке процессом записи журнала **LGWR** (**Log Writer**). Однако с ростом размеров баз данных и увеличением количества файлов это стало невыполнимой задачей для процесса **LGWR**. Если процессу **LGWR** надо обновлять десятки, сотни, а то и тысячи файлов, увеличивается вероятность того, что ожидающие фиксации транзакций сеансы будут ждать слишком долго. Процесс **CKPT** снимает эту задачу с процесса **LGWR**.

DBWn — запись блоков базы данных

Процесс записи блоков базы данных (**Database Block Writer** — **DBWn**) — фоновый процесс, отвечающий за запись измененных блоков на диск. Процесс **DBWn** записывает измененные блоки из буферного кеша, чтобы освободить пространство в кеше (чтобы освободить буферы для чтения других данных) или в ходе обработки контрольной точки (чтобы перенести вперед позицию в оперативном файле журнала повторного выполнения, с которой сервер **Oracle** начнет чтение при восстановлении экземпляра после сбоя). Как было описано ранее, при переключении журнальных файлов сервером **Oracle** запрашивается обработка контрольной точки. Серверу **Oracle** нужно перенести отметку контрольной точки, чтобы не было необходимости в только что заполненном оперативном файле журнала повторного выполнения. Если ему не удастся это сделать до того, как возникнет необходимость в файле журнала повторного выполнения, выдается сообщение, что обработка контрольной точки не завершена (**checkpoint not complete**), и придется ждать завершения обработки.

Как видите, производительность процесса **DBWn** может иметь принципиальное значение. Если он недостаточно быстро записывает блоки для освобождения буферов, сеансам приходится ждать события **FREE_BUFFER_WAITS**, и показатель '**Write Complete Waits**' начинает расти.

Можно сконфигурировать несколько (до десяти) процессов **DBWn** (**DBW0** ... **DBW9**). В большинстве систем работает только один процесс записи блоков базы данных, но в больших, многопроцессорных системах имеет смысл использовать несколько. Если сконфигурировано более одного процесса **DBWn**, не забудьте также увеличить значение параметра инициализации **DB_BLOCK_LRU_LATCHES**. Он определяет количество защелок *списков по давности использования*, **LRU lists** (теперь, в версии **8i**, их называют *списками количества обращений* — **touch lists**). Каждый процесс **DBWn** должен иметь собственный список. Если несколько процессов **DBWn** совместно используют один список блоков для записи на диск, они будут конфликтовать друг с другом при доступе к списку.

Обычно процесс **DBWn** использует асинхронный ввод-вывод для записи блоков на диск. При использовании асинхронного ввода-вывода процесс **DBWn** собирает пакет блоков для записи и передает его операционной системе. Процесс **DBWn** не ждет, пока ОС запишет блоки, — он собирает следующий пакет для записи. Завершив асинхронную запись, ОС уведомляет об этом процесс **DBWn**. Это позволяет процессу **DBWn** работать намного быстрее, чем при последовательном выполнении действий. В разделе "Подчиненные процессы" будет показано, как с помощью подчиненных процессов ввода-вывода можно эмулировать асинхронный ввод-вывод на платформах, где он не поддерживается.

И последнее замечание о процессе **DBWn**. Он, по определению, записывает блоки, разбросанные по всему диску, — процесс **DBWn** выполняет множество записей вразброс. В случае изменений будут изменяться разбросанные блоки индекса и блоки данных, достаточно случайно распределенные по диску. Процесс **LGWR**, напротив, выполняет в основном запись последовательных блоков в журнал повторного выполнения. Это — важное отличие и одна из причин, почему сервер Oracle имеет журнал повторного выполнения и отдельный процесс **LGWR**. Записи вразброс выполняются намного медленнее, чем последовательные записи. Имея грязные блоки в буферном кеше в **SGA** и процесс **LGWR**, записывающий большое количество последовательных блоков информации для восстановления измененных буферов, можно повысить производительность. Сочетание работы двух процессов — процесс **DBWn** медленно работает в фоновом режиме, тогда как процесс **LGWR** быстро выполняет работу для ожидающего пользователя — позволяет повысить общую производительность. Это верно даже несмотря на то, что сервер **Oracle** может выполнять больший объем ввода-вывода, чем надо (записывает в журнал и в файл данных), — записи в оперативный журнал повторного выполнения можно пропустить, если в ходе обработки контрольной точки сервер **Oracle** уже записал измененные блоки на диск.

LGWR — запись журнала

Процесс **LGWR** отвечает за сброс на диск содержимого буфера журнала повторного выполнения, находящегося в области **SGA**. Он делает это:

- раз в три секунды;
- при фиксации транзакции;

- при заполнении буфера журнала повторного выполнения на треть или при записи в него 1 Мбайта данных.

Поэтому создание слишком большого буфера журнала повторного выполнения не имеет смысла: сервер **Oracle** никогда не сможет использовать его целиком. Все журналы записываются последовательно, а не вразброс, как вынужден выполнять ввод-вывод процесс **DBWn**. Запись большими пакетами, как в этом случае, намного эффективнее, чем запись множества отдельных блоков в разные части файла. Это одна из главных причин выделения процесса **LGWR** и журнала повторного выполнения. Эффективность последовательной записи измененных байтов перевешивает расход ресурсов на дополнительный ввод-вывод. Сервер Oracle мог бы записывать блоки данных непосредственно на диск при фиксации, но это потребовало бы записи множества разбросанных блоков, а это существенно медленнее, чем последовательная запись изменений процессом **LGWR**.

ARCn — архивирование

Задача процесса **ARCn** — копировать в другое место оперативный файл журнала повторного выполнения, когда он заполняется процессом **LGWR**. Эти архивные файлы журнала повторного выполнения затем можно использовать для восстановления носителя. Тогда как оперативный журнал повторного выполнения используется для "исправления" файлов данных в случае сбоя питания (когда прекращается работа экземпляра), архивные журналы повторного выполнения используются для восстановления файлов данных в случае сбоя диска. Если будет потерян диск, содержащий файл данных **/d01/oradata/ora8i/system.dbf**, можно взять резервные копии за прошлую неделю, восстановить из них старую копию файла и попросить сервер применить оперативный журнал повторного выполнения и все архивные журналы, сгенерированные с момента создания этой резервной копии. Это "подтянет" файл по времени к остальным файлам в базе данных, и можно будет продолжить работу без потери данных.

Процесс **ARCn** обычно копирует оперативный журнал повторного выполнения в несколько мест (избыточность — гарантия сохранности данных!). Это могут быть диски на локальной машине или, что лучше, на другой машине, на случай катастрофического сбоя. Во многих случаях архивные файлы журнала повторного выполнения копируются затем другим процессом на третье устройство хранения, например на ленту. Они также могут отправляться на другую машину для применения к резервной базе данных (это одно из средств защиты от сбоев, предлагаемое **Oracle**).

BSP — сервер блоков

Этот процесс используется исключительно в среде **Oracle Parallel Server** (**OPS**). **OPS** — конфигурация **Oracle**, при которой несколько экземпляров монтируют и открывают одну и ту же базу данных. Каждый экземпляр **Oracle** в этом случае работает на своей машине в кластере, и все они имеют доступ для чтения и записи к одному и тому же набору файлов базы данных.

При этом буферные кеши в **SGA** экземпляров должны поддерживаться в согласованном состоянии. Для этого и предназначен процесс **BSP**. В ранних версиях OPS согласование достигалось с помощью выгрузки блока из кеша ('**ping**'). Если машине в кластере требовалось согласованное по чтению представление блока данных, заблокированного в исключительном режиме другой машиной, выполнялся обмен данными с помощью сброса на диск. В результате получалась очень дорогостоящая операция чтения данных. Сейчас, при наличии процесса **BSP**, обмен происходит из кеша в кеш через высокоскоростное соединение машин в кластере.

LMON — контроль блокировок

Этот процесс используется исключительно в среде **OPS**. Процесс **LMON** контролирует все экземпляры кластера для выявления сбоя экземпляра. Затем он вместе с диспетчером распределенных блокировок (**Distributed Lock Manager** — **DLM**), используемым аппаратным обеспечением кластера, восстанавливает глобальные блокировки, которые удерживаются сбойным экземпляром.

LMD — демон диспетчера блокировок

Этот процесс используется исключительно в среде **OPS**. Процесс **LMD** управляет глобальными блокировками и глобальными ресурсами для буферного кеша в кластерной среде. Другие экземпляры посылают локальному процессу **LMD** запросы с требованием снять блокировку или определить, кто ее установил. Процесс **LMD** также выявляет и снимает глобальные взаимные блокировки.

LCKn — блокирование

Процесс **LCKn** используется исключительно в среде **OPS**. Он подобен по функциям описанному выше процессу **LMD**, но обрабатывает запросы ко всем остальным глобальным ресурсам, кроме буферного кеша.

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.

Служебных фоновых процессов — два. Один из них запускает посланные на выполнение задания. В СУБД **Oracle** встроена очередь пакетных заданий, позволяющая выполнять по расписанию однократные или периодические задания. Другой процесс поддерживает и обрабатывает таблицы очереди, используемые средствами расширенной поддержки очередей (**Advanced Queuing** — **AQ**). Средства **AQ** обеспечивают встроенные возможности обмена сообщениями между сеансами базы данных.

SNPn — обработка снимков (очереди заданий)

Сейчас можно сказать, что имя для процесса **SNPn** выбрано неудачно. В версии 7.0 сервера **Oracle** впервые появилась поддержка репликации. Это делалось с помощью объекта базы данных, известного как моментальный снимок (snapshot). Внутренним механизмом для обновления или приведения к текущему состоянию моментальных снимков был **SNPn** — процесс обработки снимков (snapshot process). Этот процесс контролировал таблицу заданий, по которой определял, когда необходимо обновлять моментальные снимки в системе. В Oracle 7.1 корпорация Oracle открыла это средство для общего доступа через пакет **DBMS_JOB**. То, что было связано с моментальными снимками в версии 7.0, стало "очередью заданий" в версии 7.1 и последующих. Со временем имена параметров для управления очередью (как часто ее надо проверять и сколько процессов может быть в очереди) изменились со **SNAPSHOT_REFRESH_INTERVAL** и **SNAPSHOT_REFRESH_PROCESSES** на **JOB_QUEUE_INTERVAL** и **JOB_QUEUE_PROCESSES**. А вот имя процесса операционной системы не изменилось.

Можно запускать до 36 процессов очереди заданий. Они именуются **SNP0**, **SNP1**, ..., **SNP9**, **SNPA**, ..., **SNPZ**. Эти процессы очередей заданий интенсивно используются при репликации в ходе обновления моментального снимка или материализованного представления. Разработчики также часто используют их для запуска отдельных (фоновых) или периодически выполняющихся заданий. Например, далее в книге будет показано, как использовать очереди заданий для существенного ускорения обработки: за счет дополнительной работы можно сделать намного приятнее среду для пользователя (аналогично тому, как сделано в самом сервере **Oracle** при использовании процессов **LGWR** и **DBWn**).

QMNn — контроль очередей

Процесс **QMNn** по отношению к таблицам **AQ** выполняет ту же роль, что и процесс **SNPn** по отношению к таблице заданий. Этот процесс контролирует очереди и уведомляет ожидающие сообщений процессы о том, что доступно сообщение. Он также отвечает за распространение очередей — возможность переместить сообщение, поставленное в очередь в одной базе данных, в другую базу данных для извлечения из очереди.

Монитор очередей — это необязательный фоновый процесс. Параметр инициализации **AQ_TM_PROCESS** позволяет создать до десяти таких процессов с именами **QMN0**, ..., **QMN9**. По умолчанию процессы **QMNn** не запускаются.

EMNn — монитор событий

Процессы **EMNn** — часть подсистемы расширенной поддержки очередей. Они используются для уведомления подписчиков очереди о сообщениях, в которых они могут быть заинтересованы. Это уведомление выполняется асинхронно. Имеются функции **Oracle Call Interface** (**OCI**) для регистрации обратного вызова, уведомляющего о сообщении. ***Обратный вызов*** (**callback**) — это функция в программе **OCI**, которая вызывается автоматически при появлении в очереди определенного сообщения. Фоновый процесс **EMNn** используется для уведомления подписчика. Процесс **EMNn** запускается автоматически при выдаче первого уведомления в экземпляре. После этого приложение может явно вызвать **message_receive(dequeue)** для извлечения сообщения из очереди.

Подчиненные процессы

Теперь мы готовы рассмотреть последний класс процессов **Oracle** — 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

Подчиненные процессы ввода-вывода используются для эмуляции асинхронного ввода-вывода в системах или на устройствах, которые его не поддерживают. Например, ленточные устройства (чрезвычайно медленно работающие) не поддерживают асинхронный ввод-вывод. Используя подчиненные процессы ввода-вывода, можно сымитировать для ленточных устройств такой способ работы, который операционная система обычно обеспечивает для дисков. Как и в случае действительно асинхронного ввода-вывода, процесс, записывающий на устройство, накапливает большой объем данных в виде пакета и отправляет их на запись. Об их успешной записи процесс (на этот раз — подчиненный процесс ввода-вывода, а не ОС) сигнализирует исходному вызвавшему процессу, который удаляет этот пакет из списка данных, ожидающих записи. Таким образом, можно существенно повысить производительность, поскольку именно подчиненные процессы ввода-вывода ожидают завершения работы медленно работающего устройства, а вызвавший их процесс продолжает выполнять другие важные действия, собирая данные для следующей операции записи.

Подчиненные процессы ввода-вывода используются в нескольких компонентах **Oracle 8i** — процессы **DBWn** и **LGWR** используют их для имитации асинхронного ввода-вывода, а утилита **RMAN** (**Recovery MANager** — диспетчер восстановления) использует их при записи на ленту.

Использование подчиненных процессов ввода-вывода управляется двумя параметрами инициализации.

- **BACKUP_TAPE_IO_SLAVES**. Этот параметр указывает, используются ли подчиненные процессы ввода-вывода утилитой **RMAN** для резервного копирования или восстановления данных с ленты. Поскольку этот параметр предназначен для ленточных устройств, а к ленточным устройствам в каждый момент времени может обращаться только один процесс, он — булева типа, а не задает количество используемых подчиненных процессов, как можно было ожидать. Утилита **RMAN** запускает необходимое количество подчиненных процессов, в соответствии с количеством используемых физических устройств. Если параметр **BACKUP_TAPE_IO_SLAVES** имеет значение **TRUE**, то для записи или чтения с ленточного устройства используется подчиненный процесс ввода-вывода. Если этот параметр имеет (стандартное) значение **FALSE**, подчиненные процессы ввода-вывода не используются при резервном копировании. К ленточному устройству тогда обращается фоновый процесс, выполняющий резервное копирование.

- **DBWn_IO_SLAVES**. Задает количество подчиненных процессов ввода-вывода, используемых процессом **DBWn**. Процесс **DBWn** и его подчиненные процессы всегда записывают на диск измененные буфера буферного кеша. По умолчанию этот параметр имеет значение 0, и подчиненные процессы ввода-вывода не используются.

Подчиненные процессы параллельных запросов

В **Oracle 7.1** появились средства распараллеливания запросов к базе данных. Речь идет о возможности создавать для **SQL-операторов** типа **SELECT**, **CREATE TABLE**, **CREATE INDEX**, **UPDATE** и т.д. план выполнения, состоящий из **нескольких** планов, которые можно выполнять одновременно. Результаты выполнения этих планов объединяются. Это позволяет выполнить операцию за меньшее время, чем при последовательном выполнении. Например, если имеется большая таблица, разбросанная по десяти различным файлам данных, 16-процессорный сервер, и необходимо выполнить к этой таблице запрос, имеет смысл разбить план выполнения этого запроса на 16 небольших частей и полностью использовать возможности сервера. Это принципиально отличается от использования одного процесса для последовательного чтения и обработки всех данных.

3.4. Блокирование и совместный доступ

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. Механизмы блокирования и управления одновременным доступом, позволяющие решить эту проблему, являются ключевыми в любой базе данных, и в СУБД **Oracle** они весьма эффективны. Однако реализация этих механизмов в **Oracle** уникальна, и разработчик приложений должен обеспечить их корректное использование в программе при манипулировании данными.

В противном случае приложение будет работать не так, как предполагалось, и целостность данных может быть нарушена.

Подробно рассмотрим, как сервер **Oracle** блокирует данные, а также последствия выбора модели блокирования, которые необходимо учитывать при разработке многопользовательских приложений. Мы изучим уровни блокирования данных в **Oracle**, реализацию многовариантной согласованности по чтению и ее последствия для разработчиков приложений.

Блокировка — это механизм, используемый для управления одновременным доступом к общему ресурсу. Обратите внимание: использован термин "общий ресурс", а не "строка таблицы".

Сервер **Oracle** действительно 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.

Блокировки используются в базе данных для одновременного доступа к общим ресурсам и обеспечения при этом целостности и согласованности данных.

В однопользовательской базе данных блокировки не нужны. У них по определению только один пользователь, изменяющий информацию. Однако если данные или структуры данных читаются и изменяются несколькими пользователями, важно иметь штатный механизм предотвращения одновременных изменений одного и того же фрагмента информации. Именно для этого и используется блокирование. Очень важно понять, что способов блокирования в базе данных столько же, сколько и СУБД. Богатый опыт работы с моделью блокирования конкретной реляционной СУБД еще не означает, что вы знаете о блокировании все.

Разница между тем, как это делается в **Oracle** и в других СУБД, — феноменальна. Не стоит и говорить, что ни **Informix**, ни **Sybase** не пытались регистрировать результаты в базе данных в ходе тестирования. Они предпочли записывать результаты в обычные файлы операционной системы.

Из этой истории можно сделать два вывода: все СУБД существенно различаются, и при разработке приложения необходимо подходить к каждой СУБД так, будто она для вас — первая. То, что принято делать в одной СУБД, может оказаться ненужным или просто не работать в другой.

Работая с **Oracle**, вы поймете, что:

- Транзакции — это хорошо, именно для их поддержки и создавались СУБД.
- Откладывать фиксацию транзакции можно настолько, насколько необходимо. Не надо стремиться к коротким транзакциям с целью снижения нагрузки на систему, поскольку длинные или большие по объему изменений транзакции не нагружают систему. Правило следующее: фиксируйте транзакцию тогда, когда это необходимо, и не раньше. Размер транзакций диктуется только бизнес-логикой.
- Удерживать блокировки данных можно столько, сколько необходимо. Для вас это — средство, а не проблема, которой надо избегать. Блокировки не являются ограниченным ресурсом.
- Блокирование на уровне строк в **Oracle** не приводит к дополнительным расходам ресурсов.
- Никогда не нужно повышать уровень блокировки (например, блокировать таблицу вместо блокирования строк), поскольку "так лучше для системы". В **Oracle** для системы так лучше не будет: ресурсы при этом не экономятся.
- Одновременный доступ к данным и их согласованность не противоречат друг другу. Всегда можно получить результаты быстро, и при том корректные.

Проблемы блокирования

Прежде чем описывать различные типы блокировок, используемые СУБД **Oracle**, имеет смысл разобраться с рядом проблем блокирования, многие из которых возникают из-за неправильно спроектированных приложений, некорректно использующих (или вообще не использующих) механизмы блокирования базы данных.

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.

Когда получившееся в результате приложение ведет себя, как описано выше, пользователи полностью перестают ему доверять, особенно потому что подобные результаты кажутся случайными и спорадическими и абсолютно невоспроизводимы в управляемой среде тестирования (что приводит разработчика к мысли, что это, возможно, ошибка пользователя).

Многие инструментальные средства, например **Oracle Forms**, автоматически защищают разработчиков от таких ситуаций, проверяя, не изменилась ли запись с момента запроса и заблокирована ли перед началом изменений. Но другие средства разработки (и обычные программы на языке **VB** или **Java**) такой защиты не обеспечивают. 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 кране, например, когда пользователь выбирает определенную строку с целью изменения (допустим, щелкая на кнопке в окне). Итак, пользователь запрашивает данные без блокирования:

```
> 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:

```
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:

```
1 UPDATE EMP

2 SET ENAME = :ENAME, SAL = :SAL

3 WHERE EMPNO = .EMPNO;

1 row updated.

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. Должны ли мы заставлять пользователя повторно выполнять транзакцию после запроса новых значений для строки (сбивая его с толку, поскольку снова может получиться так, что изменяемая им строка обновлена другим сеансом)? Стоит ли попытаться совместить изменения, разрешая

конфликты двух изменений на основе бизнес-правил (что требует написания большого объема кода)? Конечно, для отключившихся пользователей последний вариант —единственно возможный.

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.

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.

Блокирование происходит, когда один сеанс удерживает ресурс, запрашиваемый другим сеансом. В результате запрашивающий сеанс будет заблокирован —он "повиснет" до тех пор, пока удерживающий сеанс не завершит работу с ресурсом. Блокирования практически всегда можно избежать. Если оказывается, что интерактивное приложение заблокировано, проблема, скорее всего, связана с ошибкой, подобной описанному выше потерянному изменению (логика работы приложения ошибочна, что и приводит к блокированию).

Блокирование в базе данных выполняют четыре основных оператора **ЯМД** (язык манипуляции данными): **INSERT**, **UPDATE**, **DELETE** и **SELECT FOR UPDATE**. Решение проблемы в последнем случае тривиально: добавьте конструкцию **NOWAIT**, и оператор **SELECT FOR UPDATE** больше не будет заблокирован. Вместо этого приложение должно сообщать пользователю, что строка уже заблокирована. Интерес представляют остальные три оператора **ЯМД**.

Мы рассмотрим каждый из них и увидим, почему они не должны блокировать друг друга и как это исправить, если блокирование все-таки происходит.

Заблокированные вставки

Единственный случай блокирования операторами **INSERT** друг друга, —когда имеется таблица с первичным ключом или ограничением уникальности и два сеанса одновременно пытаются вставить строку с одним и тем же значением. Один из сеансов будет заблокирован, пока другой не зафиксирует изменение (в этом случае заблокированный сеанс получит сообщение об ошибке, связанной с дублированием значения) или не откатит его (в этом случае операция заблокированного сеанса будет выполнена успешно).

Такое обычно происходит с приложениями, позволяющими генерировать пользователю первичные ключи или значения уникальных столбцов. Этой проблемы проще всего избежать за счет использования при генерации первичных ключей последовательностей **Oracle** —средства генерации уникальных ключей, обеспечивающего максимальный параллелизм в многопользовательской среде. Если нельзя использовать последовательность, можно применить метод, описанный в приложении А при рассмотрении пакета **DBMS_LOCK**. Там я демонстрирую, как решить эту проблему, прибегнув к явному блокированию вручную.

Заблокированные изменения и удаления

В интерактивном приложении, которое запрашивает данные из базы, позволяет пользователю манипулировать ими, а затем возвращает их в базу данных, заблокированные операторы **UPDATE** или

DELETE показывают, что в коде может быть проблема потерянного изменения. Вы пытаетесь изменить с помощью **UPDATE** строку, которую уже изменяет другой пользователь, другими словами, которая уже кем-то заблокирована. Этого блокирования можно избежать с помощью запроса **SELECT FOR UPDATE NOWAIT**, позволяющего:

- 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.

Взаимные блокировки

Взаимные блокировки возникают, когда два сеанса удерживают ресурсы, необходимые другому сеансу. Взаимную блокировку легко продемонстрировать на примере базы данных с двумя таблицами, **A** и **B**, в каждой из которых по одной строке. Для этого необходимо начать два сеанса (скажем, два сеанса **SQL*Plus**) и в сеансе **A** изменить таблицу **A**. В сеансе **Б** надо изменить таблицу **B**. Теперь, если попытаться изменить таблицу **A** в сеансе **Б**, он окажется заблокированным, поскольку соответствующая строка уже заблокирована сеансом **A**. Это еще не взаимная блокировка —сеанс просто заблокирован. Взаимная блокировка еще не возникла, поскольку есть шанс, что сеанс **A** зафиксирует или откатит транзакцию и после этого сеанс **Б** продолжит работу.

Если мы вернемся в сеанс **A** и попытаемся изменить таблицу **B**, 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 **A** 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 **A** изменить таблицу **B** 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 **B**, а сеанс **A** терпеливо ждет, пока эта строка станет доступной. Получив сообщение о взаимной блокировке, сеанс **Б** должен решить, фиксировать ли уже

выполненные изменения в таблице **B**, 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. Содержимое трассировочного файла примерно таково:

```
*** 2001-02-23 14:03:35.041

*** SESSION ID:(8.82) 2001-02-23 14:03:35.001

DEADLOCK DETECTED

Current SQL statement for this session:

update a set x = x+1
```

The following deadlock is not an ORACLE error. It is a

deadlock due to user error in the design of an application

or from issuing incorrect ad-hoc SQL. The following...

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).

Блокирование таблицы

С 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 С, 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. Если кто-то жалуется на взаимные блокировки в базе данных, я предлагаю выполнить сценарий, который находит неиндексированные внешние ключи, и в девяноста девяти процентах случаев мы обнаруживаем таблицу, вызвавшую проблемы.

После индексирования соответствующего внешнего ключа взаимные блокировки и множество других конфликтов при доступе исчезают навсегда.

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). Специфические блокировки **OPS** мы рассмотрим лишь вкратце, зато общие блокировки — очень подробно.

- Блокировки **ЯМД** (**DML locks**). **ЯМД** означает язык манипулирования данными (**Data Manipulation Language**), т.е. операторы **SELECT**, **INSERT**, **UPDATE** и **DELETE**. К блокировкам ЯМД относятся, например, блокировки строки данных или блокировка на уровне таблицы, затрагивающая все строки таблицы.
- Блокировки **ЯОД** (**DDL locks**). **ЯОД** означает язык определения данных (**Data Definition Language**), т.е. 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.

Блокировки **ЯМД** 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**.

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). Иногда при обращении данные кажутся заблокированными, хотя фактически они уже 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 как бы 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 помещен в очередь в ожидании завершения транзакции, после чего получит возможность работать с данными.

Блокировки ЯОД

Блокировки ЯОД автоматически устанавливаются на объекты в ходе выполнения операторов **ЯОД** для защиты их от изменения другими сеансами. Например, при выполнении оператора **ЯОД ALTER TABLE T** на таблицу **T** будет установлена исключительная блокировка **ЯОД**, что предотвращает установку блокировок **ЯОД** и **TM** на эту таблицу другими сеансами. Блокировки **ЯОД** удерживаются на период выполнения оператора **ЯОД** и снимаются сразу по его завершении. Это делается путем помещения операторов **ЯОД** в неявные пары операторов фиксации (или фиксации и отката). Вот почему операторы **ЯОД** в **Oracle** всегда фиксируются. Операторы **CREATE, ALTER** и т.д.

фактически выполняются, как показано в следующем псевдокоде:

```
Begin
Commit;
Оператор ЯОД
Commit;
Exception
When others then rollback;
End;
```

Поэтому операторы **ЯОД** всегда фиксируют транзакцию, даже если завершаются неудачно. Выполнение оператора **ЯОД** начинается с фиксации. Помните об этом. Сначала выполняется фиксация, чтобы в случае отката не пришлось откатывать предыдущую часть транзакции. При выполнении оператора **ЯОД** фиксируются все выполненные ранее изменения, даже если сам оператор **ЯОД** выполнен неудачно. Если должен быть выполнен оператор **ЯОД**, но не требуется, чтобы он зафиксировал существующую транзакцию, можно использовать автономную транзакцию.

Имеется три типа блокировок **ЯОД**:

- Исключительные блокировки **ЯОД**. Они предотвращают установку блокировок **ЯОД** или **TM** (**ЯМД**) другими сеансами. Это означает, что можно запрашивать таблицу в ходе выполнения оператора **ЯОД**, но нельзя ее изменять.
- Разделяемые блокировки **ЯОД**. 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 **ЯОД**, 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 **ЯОД**.

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 **ЯОД**, блокируются. В **Oracle 8i** некоторые операторы **ЯОД** теперь могут выполняться без установки блокировок **ЯОД**. Например, можно выполнить:

```
create index t_idx on t(x) ONLINE;
```

Ключевое слово **ONLINE** изменяет метод построения индекса. Вместо установки исключительной блокировки **ЯОД**, предотвращающей изменения данных, **Oracle** попытается установить на таблицу низкоуровневую (режим 2) блокировку **TM**. Это предотвращает изменения структуры с помощью операторов **ЯОД**, но позволяет нормально выполнять операторы **ЯМД**. Сервер **Oracle** достигает этого путем записи в таблице изменений, сделанных в ходе выполнения оператора **ЯОД**, и учитывает эти изменения в новом индексе, когда завершается его создание. Это существенно увеличивает доступность данных.

Другие типы операторов **ЯОД** устанавливают разделяемые блокировки **ЯОД**. 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;
```

Блокирование и одновременный доступ разделяемые блокировки **ЯОД** 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 **ЯОД** — **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.

При поиске этой информации особо ценным будет представление **DBA_DDL_LOCKS** (ни одного походящего для этого представления **V$** не существует). Представление **DBA_DDL_LOCKS** строится по более "мистическим" таблицам **X$** и по умолчанию не создается в базе данных. Для установки его и других представлений, связанных с блокировками, выполните сценарий **CATBLOCK.SQL** из каталога **[ORACLE_HOME]/rdbms/admin**. Этот сценарий можно успешно выполнить от имени пользователя **SYS**.

После выполнения этого сценария можно выполнять запросы к указанному представлению.

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 (как фактически и применение к ней большинства операторов ЯОД) другими сеансами. В изменяемых блоках оставляют блокировки **TX** —благодаря этому другие сеансы "знают", что с данными работают. Сервер использует блокировки **ЯОД** 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.

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**. Может показаться странным, зачем вообще создавать собственные блокировки. Ответ обычно зависит от того, какое используется приложение.

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.