Транзакции

Транзакции — это одно из средств, отличающих базу данных от файловой системы. Если вы находитесь в процессе записи файла и операционная система терпит крах, этот файл, скорее всего, будет поврежден. Правда, существуют файловые системы с ведением журналов, которые могут восстановить ваш файл до некоторого момента времени. Однако если вам нужно обеспечить синхронизацию двух файлов, то система не может помочь вами в этом — если вы обновите один файл, и в системе произойдет сбой прежде, чем вы завершите обновление второго, то вы получите несинхронизированные файлы. В том и состоит основное назначение транзакций базы данных — они переводят базу из одного согласованного состояния в другое. В этом заключается их работа. Когда вы фиксируете работу в базе данных, то можете быть уверены, что все разнообразные правила и проверки, защищающие целостность базы, реализованы. В предыдущей главе мы говорили о транзакциях в терминах управления параллелизмом и о том, как в результате использования **многоверсионной** согласованной по чтению модели **Oracle** транзакции **Oracle** могут обеспечить согласованность данных в любое время, в условиях высоко конкурентного доступа к данным. Транзакции в **Oracle** удовлетворяют всем требуемым характеристикам **ACID**. Аббревиатура **ACID** означает:

- атомарность (**atomicity**) — выполняется либо вся транзакция целиком, либо она целиком не выполняется;
- согласованность (**consistency**) — транзакция переводит базу данных из одного согласованного состояния в другое;
- изоляция (**isolation**) — эффект от транзакции не виден другим транзакциям до тех пор, пока она не будет зафиксирована;
- устойчивость (**durability**) — как только транзакция зафиксирована, она остается постоянной.

Операторы управления транзакциями

В **Oracle** нет необходимости в операторе “начала транзакции”. Транзакция начинается неявно с первым же оператором, который модифицирует данные (первый оператор, получающий блокировку **TX**). Вы можете явно начать транзакцию, используя команду **SET TRANSACTION** или пакет **DBMS_TRANSACTION**, но этот шаг в **Oracle** не обязателен, в отличие от различных прочих СУБД. Оператор **COMMIT** или **ROLLBACK** явно завершает транзакцию. Вы всегда должны явно завершать транзакцию с помощью **COMMIT** или **ROLLBACK**, в противном случае инструмент/среда, используемая вами, самостоятельно выберет то или другое за вас. Если вы нормально выходите их сеанса **SQL*Plus**, без фиксации или отката, то **SQL*Plus** предполагает, что вы хотите зафиксировать вашу работу и делает это. С другой стороны, если вы выходите из программы **Pro*C**, то имеет место неявный откат транзакции. Никогда не полагайтесь на неявное поведение, потому что в будущем оно может измениться. Всегда завершайте явно свои транзакции командами **COMMIT** или **ROLLBACK**. Транзакции в **Oracle атомарны** — в том смысле, что каждый оператор, составляющий транзакцию, фиксируется (становится постоянным) или же происходит откат их всех. Эта защита распространяется также и на индивидуальные операторы. Весь оператор либо полностью успешен, либо полностью неудачен. Обратите внимание, что я сказал, что “оператор” откатывается. Сбой одного оператора не вызывает автоматического отката всех предыдущих операторов. Их работа предохраняется и должна быть либо зафиксирована, либо отменена (путем отката) вами. Прежде чем мы обратимся к деталям относительно того, что означает атомарность для оператора и транзакции, давайте рассмотрим различные операторы управления транзакциями, которые доступны нам:

- **COMMIT**. Чтобы использовать простейшую форму этого оператора, вы просто пишете **COMMIT**. Можно применить более многословную форму и записать **COMMIT WORK**, но эти две формы эквивалентны. **COMMIT** завершает вашу транзакцию и фиксирует все проведенные в ней изменения как постоянные. Существуют расширения оператора **COMMIT** для распределенных транзакций. Эти расширения позволяют вам пометить **COMMIT** некоторым осмысленным комментарием и принудительно зафиксировать распределенную транзакцию.
- **ROLLBACK**. Чтобы использовать простейшую форму этого оператора, вы просто пишете **ROLLBACK**. Опять же, существует более многословный вариант **ROLLBACK WORK**, но эти формы эквивалентны. Откат (**rollback**) завершает вашу транзакцию и отменяет все незафиксированные изменения, проведенные в ней. Это делается посредством чтения информации, хранящейся в сегментах отката/отмены (я пойду дальше, и стану называть их сегментами отмены (**undo segments**), как это принято в терминологии **Oracle 10g**) и восстановления блоков базы данных в состояние, в котором они пребывали до начала транзакции.
- **SAVEPOINT**. **SAVEPOINT** позволяет вам создать “маркерную точку” внутри транзакции. В одной транзакции можно иметь множество таких точек. **ROLLBACK TO <SAVEPOINT>** оператор применяется вместе с командой **SAVEPOINT**. Вы можете откатить транзакцию к этой маркерной точке, не откатывая работу, выполненную до нее. Таким образом, вы можете написать два оператора **UPDATE**, за которыми следует **SAVEPOINT**, и затем — два оператора **DELETE**. Если ошибка или любого рода исключительная ситуация случится во время выполнения операторов **DELETE**, вы можете перехватить исключение и выполнить команду **ROLLBACK TO SAVEPOINT**

— транзакция откатится к именованной **SAVEPOINT**, отменяя всю работу, выполненную операторами **DELETE**, но оставляя в силе то, что сделано операторами **UPDATE**.

- **SET TRANSACTION**. Этот оператор позволяет устанавливать различные транзакционные атрибуты, такие как уровень изоляции транзакции и указание на то, является ли она транзакцией только для чтения или для чтения-записи. Вы можете также применять этот оператор для того, чтобы заставить транзакцию использовать специфический сегмент отката при ручном управлении отменой, хотя это и не рекомендуется. Автоматическое и ручное управление отменой мы рассмотрим более подробно в главе 9. Вот и все, больше нет никаких других операторов управления транзакциями. Наиболее часто используемые управляющие операторы — это **COMMIT** и **ROLLBACK**. Оператор **SAVEPOINT** имеет довольно-таки специальное назначение. Внутренне Oracle использует его часто, и вы также можете найти ему применение в своих приложениях.

Ограничения целостности и транзакции

Интересно отметить точно, когда проверяются ограничения целостности. По умолчанию ограничения целостности проверяются после обработки всего оператора **SQL**. Есть также отложенные ограничения, которые допускают перенос верификации ограничений целостности на тот момент, когда приложение явно запросит верификацию командой **SET CONSTRAINTS ALL IMMEDIATE** или выдав **COMMIT**.

Ограничения **IMMEDIATE** Для первой части дискуссии предположим, что ограничения находятся в режиме **IMMEDIATE**, что является нормой. В этом случае ограничения целостности проверяются немедленно после обработки всего оператора **SQL**. Обратите внимание, что я использую термин “**оператор SQL**”, а не просто “оператор”. Если у меня есть много операторов **SQL** в хранимой процедуре **PL/SQL**, то за исполнением каждого из них будет немедленно следовать верификация ограничений целостности, а не после того, как завершится вся процедура. Так почему же ограничения проверяются после выполнения оператора **SQL**? Почему не во время? Дело в том, что для отдельных операторов очень естественно делать отдельные строки таблицы на мгновение “несогласованными”. Обращение к результату частичной работы оператора может заставить **Oracle** отклонить такой результат, даже если общий результат всей работы будет корректным. Например, предположим, что у нас есть следующая таблица:

```
ops$tkyte@ORA10G> create table t ( x int unique );

Table created. Таблица создана. ops$tkyte@ORA10G> insert into t values ( 1 );

1 row created.

1 строка создана.

ops$tkyte@ORA10G> insert into t values ( 2 );

1 row created.

1 строка создана.
```

И мы хотим выполнить многострочный **UPDATE**:

```
ops$tkyte@ORA10G> update t set x = x+1;

2 rows updated.

2 строки создано.
```

Если **Oracle** проверит ограничение после обновления каждой строки, то в любой день существует вероятность 50/50 того, что **UPDATE** провалится. Строки в T доступны в некотором порядке, и если **Oracle** обновит строку X=1 первой, то если получается мгновенное дублированное значение X, оператор UPDATE будет отвергнут. Но поскольку **Oracle** терпеливо ожидает конца работы оператора, он завершается успешно, поскольку на момент его завершения дубликатов уже нет. Ограничения **DEFERRABLE** и каскадные обновления Начиная с **Oracle 8.0**, мы также имеем возможность отложить проверку ограничений, что может оказаться довольно-таки выгодно для различных операций. Первое, что немедленно приходит в голову — это требование каскадного оператора **UPDATE** первичного ключа и его дочерних ключей. Многие люди возразят, что этого делать никогда не нужно, поскольку первичные

ключи неизменны (я тоже принадлежу к их числу). Но многие другие настаивают на своем желании использовать каскадные **UPDATE**. Отложенные ограничения предоставляют такую возможность.

В предшествующих версиях можно было выполнять **CASCADE UPDATE**, но эта операция требует огромного объема работы и имеет определенные пределы. С отложенными ограничениями задача становится почти тривиальной. Код может выглядеть так, как показано ниже.

```
ops$tkyte@ORA10G> create table p
  2 ( pk int primary key )
  3 / Table created.
  Таблица создана.

ops$tkyte@ORA10G> create table c
  2 ( fk constraint c_fk
  3     references p(pk)
  4     deferrable
  5     initially immediate
  6 )
  7 / Table created.
  Таблица создана.

ops$tkyte@ORA10G> insert into p values ( 1 );
1 row created.
1 строка создана.

ops$tkyte@ORA10G> insert into c values ( 1 );
1 row created.
1 строка создана.
```

У нас есть родительская таблица **P** и дочерняя таблица **C**. Таблица **C** ссылается на таблицу **P**, и ограничение, используемое для обеспечения этого правила, называется **C_FK** (**child foreign key** — дочерний внешний ключ). Ограничение описано как **DEFERRABLE**, но установлено в INITIALLY **IMMEDIATE**. Это значит, что мы можем отложить проверку ограничения до **COMMIT** или до некоторого другого момента. Однако по умолчанию оно будет проверяться на уровне оператора. Это наиболее распространенный случай применения отложенных ограничений. Большинство существующих приложений не проверяют нарушение ограничений по оператору **COMMIT**, и лучше не менять этого. Согласно определению таблицы С, она ведет себя таким же образом, как всегда должны вести себя таблицы, но при этом предоставляет нам возможность изменить ее поведение. Попробуем воспользоваться некоторым DML для таблиц и посмотрим, что произойдет:

```
ops$tkyte@ORA10G> update p set pk = 2;
```

```
update p set pk = 2
*
ERROR at line 1:
ORA-02292: integrity constraint (OPS$TKYTE.C_FK) violated - child record found
ОШИБКА в строке 1:
ORA-02292: нарушение ограничения целостности (OPS$TKYTE.C_FK) — найдена дочерняя запись
```

Поскольку ограничение находится в режиме **IMMEDIATE**, оператор **UPDATE** завершился неудачей. Изменим режим и попробуем снова:

```
ops$tkyte@ORA10G> set constraint c_fk deferred;
Constraint set.
Ограничение установлено.
ops$tkyte@ORA10G> update p set pk = 2;
1 row updated.
1 строка обновлена.
```

Распределенные транзакции

Одним из действительно полезных свойств СУБД **Oracle** является ее способность прозрачно обрабатывать распределенные транзакции. Можно обновить данные во множестве разных баз данных за одну транзакцию. Когда вы выполняете ее фиксацию, то либо фиксируются все обновления во всех экземплярах, либо не фиксируется ни одно из них (все они откатываются). Вам не нужен для этого никакой дополнительный код, а просто оператор **commit**.

Ключом к распределенным транзакциям в **Oracle** является связь баз данных (**database link**). Эта связь представляет собой объект базы данных, описывающий способ регистрации в другом экземпляре базы из вашего экземпляра. Как только вы настроите связь баз данных, доступ к удаленным объектам становится очень простым:

```
select * from T@another_database;
```

Это позволит выбрать данные из таблицы **T** в экземпляре базы данных, определенном связью **ANOTHER_DATABASE**. Обычно вы должны “скрывать” тот факт, что **T** — удаленная таблица, создавая ее представление или синоним. Например, можно выполнить следующую команду и затем обращаться к **T**, как если бы это была локальная таблица:

```
create synonym T for T@another_database;
```

Теперь, имея настроенную связь с удаленной базой и получив возможность читать некоторые таблицы, вы также можете модифицировать их (конечно, при условии обладания соответствующими привилегиями). Выполнение распределенной транзакции теперь ничем не отличается от транзакции локальной. Вот все, что для этого нужно:

```
update local_table set x = 5;
update remote_table@another_database set y = 10;
```

```
commit;
```

Здесь СУБД **Oracle** выполнит фиксацию либо в обеих базах данных, либо ни в одной из них. Она использует протокол **2PC** (**двухфазной фиксации**). Этот протокол позволяет выполнять модификации, затрагивающие множество различных баз данных, фиксируя их автоматически. Он, насколько возможно, пытается перекрыть все пути для распределенных сбоев перед тем, как выполнить фиксацию. В **2PC** между многими базами данных одна из баз — обычно та, к которой клиент подключился изначально — служит координатором распределенной транзакции. Этот сайт запросит у других сайтов готовности к фиксации. То есть этот сайт обратится к другим сайтам и попросит их подготовиться к фиксации. Каждый из этих других сайтов рапортует о своем "состоянии готовности", как "**ДА**" или "**НЕТ**". Если любой из сайтов говорит "**НЕТ**", выполняется откат всей транзакции. Если же все сайты рапортуют "**ДА**", сайт-координатор рассылает сообщение с командой на выполнение фиксации на всех сайтах.

Это ограничивает возможности для серьезных ошибок. Прежде чем выполнится "опрос" по **2PC**, любая распределенная ошибка приведет к выполнению отката на всех сайтах. Не будет никаких сомнений относительно исхода распределенной транзакции. После команды на фиксацию или откат опять-таки нет никаких сомнений относительно исхода распределенной транзакции. Лишь в течение очень короткого периода ("окна") времени, когда координатор собирает ответы, исход может быть неоднозначным после сбоя.

Предположим, например, что у нас есть три сайта, участвующих в транзакции. **Сайт 1** выступает в роли координатора. **Сайт 1** просит **сайт 2** подготовиться к фиксации, и **сайт 2** выполняет это. Затем **сайт 1** просить **сайт 3** подготовиться к фиксации, и он также делает это. В этот момент времени **сайт 1** — единственный, кто знает об исходе транзакции, и теперь он отвечает за извещение об этом других сайтов. Если ошибка случится прямо сейчас — произойдет сбой сети, **сайт 1** останется без питания или еще что-то — **сайты 2 и 3** останутся в "подвешенном" состоянии. Они получат то, что называется сомнительной распределенной транзакцией. Протокол **2PC** пытается закрыть "окно" ошибок, насколько это возможно, но не может исключить такую вероятность полностью. **Сайты 2 и 3** должны удерживать транзакцию открытой, ожидая нотификации от **сайта 1** команды на фиксацию или откат. Для разрешения этой проблемы существует процесс **RECO**. Это также тот случай, когда на сцену выходят **COMMIT** и **ROLLBACK** с опцией **FORCE**. Если причиной проблемы был сбой сети между **сайтами 1, 2 и 3**, то администраторы **сайтов 2 и 3** должны обратиться к администратору **сайта 1**, запросить его об исходе транзакции и выполнить, соответственно, фиксацию или откат вручную. Существует несколько (немного) ограничений относительно того, что вы можете делать в распределенной транзакции, и все они оправданы. Ниже перечислены наиболее существенные из них.

- Вы не можете выдать **COMMIT** по связи баз данных. То есть, нельзя дать команду **COMMIT@удаленный_сайт**. Вы можете зафиксировать транзакцию только на сайте, который ее инициировал.
- Вы не можете выполнять **DDL** в удаленной базе по связи баз данных. Это — прямое следствие предыдущего ограничения. **DDL** выполняет фиксацию. Нельзя выполнить фиксацию ни с одного сайта, кроме инициирующего, а потому нельзя выполнять **DDL** по связи баз данных.
- Нельзя выдать **SAVEPOINT** по связи баз данных. Короче говоря, вы не можете выдать ни одного оператора управления транзакциями по связи баз данных. Все управление транзакциями наследуется от сеанса, который первоначально открывает связи баз данных. Вы не можете осуществлять другого управления транзакциями на распределенных экземплярах в транзакции.

Недостаток управления транзакциями по связи баз данных также оправдан, поскольку инициирующий сайт — единственный, имеющий список всех, кто вовлечен в транзакцию. Если в нашей конфигурации из трех сайтов **сайт 2** попытается зафиксировать транзакцию, у него не будет никаких сведений о том, что в ней участвует **сайт 3**. Поэтому в **Oracle** только **сайт 1** может выдать команду на фиксацию. В этой точке для **сайта 1** допустимо делегировать ответственность за управление распределенной транзакцией другому сайту.

Мы можем указать, какой сайт будет действительным фиксирующим сайтом, устанавливая параметр **COMMIT_POINT_STRENGTH** ("сила точки фиксации") сайта. Сила точки фиксации ассоциирует относительный уровень важности с сервером в распределенной транзакции. Чем более важен сервер (больше доступных данных должно быть на нем), тем более вероятно, что именно он будет координировать распределенную транзакцию. Вам может понадобиться это в случае, когда необходимо выполнить распределенную транзакцию между вашим рабочим сервером и тестовым сервером. Поскольку координатор транзакции никогда не сомневается в исходе транзакции, будет лучше, если рабочий сервер станет координировать распределенную транзакцию. Вам не нужно особо беспокоиться о вашем тестовом сервере — что на нем вдруг окажутся открытые транзакции или блокированные ресурсы. Вам определенно стоит беспокоиться, если подобное случится на рабочем сервере.

В невозможности выполнения **DDL** по связи баз данных вообще нет ничего плохого. Во-первых, **DDL** случается редко. Вы выполняете операторы **DDL** при инсталляции или обновлении. Рабочие системы не выполняют **DDL** (по крайней мере, не должны). Во-вторых, все-таки существует способ запустить **DDL** через связь баз данных — воспользоваться средством планирования заданий **DBMS_JOB** или, в **Oracle 10g** — пакетом планировщика **DBMS_SCHEDULER**. Вместо того чтобы пытаться выполнить **DDL** по связи, вы используете связь для планирования удаленного задания, чтобы оно выполнилось, как только

вы осуществите фиксацию. Таким образом, задание запустится на удаленной машине, оно не будет частью распределенной транзакции и потому сможет выполнить операторы **DDL**. Фактически это метод, посредством которого сервер репликации **Oracle** (**Oracle Replication Server**) выполняет распределенные команды **DDL**, чтобы осуществить репликацию схемы.

Автономные транзакции

Автономные транзакции позволяют вам создать “транзакцию внутри транзакции”, которая зафиксирует или выполнит откат изменений, независимо от родительской транзакции. Они позволяют вам приостановить текущую выполняемую транзакцию, запустить новую, проделать некоторую работу и зафиксировать или откатить ее — все это не затрагивая состояния текущей выполняемой транзакции. Автономные транзакции предоставляют новый метод управления транзакциями в **PL/SQL** и могут быть использованы в следующих конструкциях.

- Анонимные блоки верхнего уровня.
- Локальные (процедура в процедуре), автономные или пакетные функции и процедуры.
- Методы объектных типов.
- Триггеры баз данных.

Прежде чем мы рассмотрим работу автономных транзакций, необходимо подчеркнуть, что это — весьма мощный, а потому опасный инструмент, если применять его неправильно. Реальная потребность в автономных транзакциях возникает очень редко. Слишком легко непреднамеренно нарушить логическую целостность данных в системе с автономными транзакциями. Лучший способ продемонстрировать действие и последствия автономных транзакций — привести пример.

Создадим простую таблицу для хранения сообщений:

```
ops$tkyte@ORA10G> create table t ( msg varchar2(25) );

Table created.

Таблица создана.
```

Далее мы создадим две процедуры, каждая из которых просто вставляет свое имя в таблицу сообщения и сразу фиксирует эту вставку. Однако одна из этих процедур будет обычной, а другая — закодированной в виде автономной транзакции. Мы используем эти объекты, чтобы показать, какая работа фиксируется в базе данных при определенных обстоятельствах. Для начала — процедура **AUTONOMOUS_INSERT**:

```
ops$tkyte@ORA10G> create or replace procedure Autonomous_Insert
  2 as
  3    pragma autonomous_transaction;
  4 begin
  5    insert into t values ( 'Autonomous Insert' );
  6    commit;
  7 end;
  8 / Procedure created.

Процедура создана.
```

Обратите внимание на использование **pragma AUTONOMOUS_TRANSACTION**. Эта директива сообщает базе данных, что процедура должна выполняться как новая автономная транзакция, независимо от ее родительской транзакции.

На заметку: **pragma** — это просто директива компилятора, то есть метод инструктирования компилятора, чтобы он выполнил некоторую опцию компиляции. Доступны и другие **pragma**. Обратитесь к руководству по **PL/SQL** и вы увидите их полный список в указателе.

А вот — “правильная” процедура **NONAUTONOMOUS_INSERT**:

```
ops$tkyte@ORA10G> create or replace procedure NonAutonomous_Insert

2 as

3 begin

4   insert into t values ( 'NonAutonomous Insert' );

5   commit;

6 end;

7 / Procedure created.

Процедура создана.
```

Теперь давайте понаблюдаем за поведением неавтономной транзакции в анонимном блоке кода **PL/SQL**:

```
ops$tkyte@ORA10G> begin

2    insert into t values ( 'Anonymous Block' );

3    NonAutonomous_Insert;

4    rollback;

5 end;

6 / PL/SQL procedure successfully completed.

Процедура PL/SQL успешно завершена.

ops$tkyte@ORA10G> select * from t;

MSG

----------------------

Anonymous Block

NonAutonomous Insert
```

Как видите, работа, выполненная анонимным блоком — его оператор **INSERT** — была зафиксирована процедурой **NONAUTONOMOUS_INSERT**. Обе строки данных были зафиксированы, так что команде **ROLLBACK** откатывать нечего. Сравните это с поведением процедуры автономной транзакции:

```
ops$tkyte@ORA10G> delete from t;
```

```
2 rows deleted.

2 строки удалено.

ops$tkyte@ORA10G> commit;

Commit complete.

Фиксация завершена.

ops$tkyte@ORA10G> begin

2   insert into t values ( 'Anonymous Block' );

3   Autonomous_Insert;

4   rollback;

5 end;

6 / PL/SQL procedure successfully completed.

Процедура PL/SQL успешно завершена.

ops$tkyte@ORA10G> select * from t;

MSG

----------

Autonomous Insert
```

Здесь сохраняется только работа, выполненная и зафиксированная в автономной транзакции. Оператор **INSERT**, выполненный в анонимном блоке, откатывается оператором **ROLLBACK** в строке 4. **COMMIT** процедуры автономной транзакции не оказывает никакого эффекта на родительскую транзакцию, которая стартовала в анонимном блоке. По сути, здесь зафиксирована сущность автономных транзакций и то, что они делают. Подводя итоги, можно сказать, что если вы выполняете **COMMIT** внутри “нормальной” процедуры, он касается не только ее собственной работы, но также всей внешней работы, выполненной в данном сеансе. Однако **COMMIT**, выполненный в процедуре с автономной транзакцией, распространяется только на работу этой процедуры.

Транзакции — одно из главных средств, отличающих базу данных от файловой системы. Понимание их работы и правильное их применение необходимо для корректной реализации приложений в любой базе данных. Очень важно понимание того, что в **Oracle** любой оператор является атомарным (включая все побочные эффекты от этого), и что атомарность распространяется и на хранимые процедуры. Мы видели, как помещение в блок **PL/SQL** обработчика исключений **WHEN OTHERS** может радикально повлиять на то, какие изменения произойдут в базе данных. Для разработчиков базы данных четкое понимание работы транзакций имеет важнейшее значение.