ГЛАВА 8

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.

В прошлой главе мы обсуждали, как Oracle обеспечивает *согласованность* и *изоляцию*. Здесь мы сосредоточим внимание на концепции *атомарности* и ее применении в СУБД Oracle.

В настоящей главе мы поговорим о влиянии атомарности и о том, как она касается операторов SQL в Oracle. Мы расскажем об операторах управления транзакциями, таких как `COMMIT`, `SAVEPOINT` и `ROLLBACK`, и обсудим способы обеспечения ограничений целостности в транзакциях. Также мы рассмотрим некоторые плохие привычки, связанные с транзакциями, которые вы могли приобрести, разраба-

тывая приложения для других СУБД. Речь пойдет также о распределенных транзакциях и двухфазной фиксации (2PC). И последняя тема, которая будет поднята здесь — это автономные транзакции: что они собой представляют и какую играют роль.

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.

На заметку! Команда `ROLLBACK TO SAVEPOINT` не завершает транзакцию! Только полная правильная команда `ROLLBACK` может ее завершить.

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 10*g*) 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>. Этот 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.

Атомарность

После этого краткого обзора операторов управления транзакциями мы готовы рассмотреть, что означает атомарность оператора, процедуры и транзакции.

Атомарность уровня оператора

Рассмотрим следующий оператор:

```
Insert into t values(1);
```

Достаточно ясно, что он должен завершиться неудачей при нарушении ограничений — строка не будет вставлена. Однако рассмотрим следующий пример, где INSERT или DELETE по таблице T возбуждает триггер, который соответственно изменяет значение столбца CNT в таблице T2:

```
ops$tkyte@ORA10G> create table t2 ( cnt int );
Table created.
Таблица создана.
```

```
ops$tkyte@ORA10G> insert into t2 values ( 0 );
1 row created.
1 строка создана.

ops$tkyte@ORA10G> commit;
Commit complete.
Фиксация завершена.

ops$tkyte@ORA10G> create table t ( x int check ( x>0 ) );
Table created.
Таблица создана.

ops$tkyte@ORA10G> create trigger t_trigger
 2 before insert or delete on t for each row
 3 begin
 4   if ( inserting ) then
 5     update t2 set cnt = cnt +1;
 6   else
 7     update t2 set cnt = cnt -1;
 8   end if;
 9   dbms_output.put_line( 'Я вставил и обновил ' ||
10                         sql%rowcount || ' строк(у)' );
11 end;
12 /
Trigger created.
Триггер создан.
```

В этой ситуации уже не так ясно, что должно случиться. Если ошибка случится *после* срабатывания триггера, должен ли эффект от триггера оставаться в силе или нет? То есть, если триггер активизирован и обновил `T2`, но строка не была вставлена в `T`, каким должен быть результат? Ясно, что мы не хотим, чтобы в этом случае значение столбца `CNT` в `T2` было увеличено, если строка не была действительно вставлена в `T`. К счастью, в Oracle исходный оператор, введенный клиентом — в этом случае `INSERT INTO T` — либо полностью проходит, либо полностью не проходит. То есть он является атомарным. Это можно подтвердить следующим образом:

```
ops$tkyte@ORA10G> set serveroutput on
ops$tkyte@ORA10G> insert into t values (1);
I fired and updated 1 rows
1 row created.
1 строка создана.

ops$tkyte@ORA10G> insert into t values(-1);
Я вставил и обновил 1 строк(у)
insert into t values(-1)
*
ERROR at line 1:
ORA-02290: check constraint (OPS$TKYTE.SYS_C009597) violated
ОШИБКА в строке 1:
ORA-02290: нарушение проверочного ограничения целостности (OPS$TKYTE.SYS_C009597)

ops$tkyte@ORA10G> select * from t2;
      CNT
---------
        1
```

На заметку! При использовании SQL*Plus из Oracle9i Release 2 и предшествующих версий для того, чтобы увидеть, что триггер сработал, вам нужно добавить строку кода `exec null` после второй вставки. Это связано с тем, что SQL*Plus в этой версии не извлекает и не отображает информацию `DBMS_OUTPUT` после неудачного оператора DML, тогда как в Oracle 10*g* — отображает.

Итак, одна строка была успешно вставлена в `T`, и мы получили надлежащее сообщение — "Я вставил и обновил 1 строк(у)". Следующий оператор `INSERT` нарушает ограничение целостности, установленное для `T`. Сообщение `DBMS_OUTPUT` свидетельствует, что триггер `T` фактически сработал, и мы видим тому подтверждение. Он успешно выполнил свои обновления `T2`. Возможно, вы ожидали, что `T2` теперь содержит значение `2`, но на самом деле мы видим, что оно равно `1`. СУБД Oracle делает *исходный* оператор `INSERT` атомарным — то есть первоначальный `INSERT INTO T` является оператором, а любой сторонний эффект этого оператора рассматривается как его часть.

СУБД Oracle достигает такой атомарности уровня оператора за счет того, что молча снабжает конструкцией `SAVEPOINT` каждый вызов в базе данных. Предыдущие два `INSERT` на самом деле трактуются так:

```
Savepoint statement1;
Insert into t values ( 1 );
If error then rollback to statement1;
Savepoint statement2;
Insert into t values ( -1 );
If error then rollback to statement2;
```

Для программистов, работавших с Sybase или SQL Server, поначалу это может показаться непонятным. В этих СУБД как раз *истинно в точности обратное*. В этих системах триггеры выполняются независимо от вызвавшего их оператора. Если возникает ошибка, триггеры должны явно откатить свою собственную работу и затем возбудить другую ошибку, чтобы откатить исходный оператор, который их инициирует. В противном случае работа, выполненная триггером, может остаться в силе, даже если вызвавший их оператор или другая его часть, в конечном счете, завершилась сбоем.

В Oracle атомарность уровня оператора распространяется так глубоко, как это должно быть. Если в предыдущем примере `INSERT INTO T` возбудит триггер, обновляющий другую таблицу, а та таблица снабжена триггером, выполняющим удаление в третьей таблице (и так далее, и тому подобное), то либо *вся* работа будет выполнена успешно, либо *никакая* выполнена не будет. Вам не нужно ничего дополнительно кодировать, чтобы обеспечить это — просто так оно работает.

Атомарность уровня процедуры

Интересно отметить, что Oracle также трактует анонимный блок PL/SQL как один оператор. Рассмотрим следующую хранимую процедуру:

```
ops$tkyte@ORA10G> create or replace procedure p
  2 as
  3 begin
  4       insert into t values ( 1 );
```

```
  5      insert into t values (-1 );
  6 end;
  7 /
Procedure created.
Процедура создана.

ops$tkyte@ORA10G> select * from t;
no rows selected
нет выбранных строк

ops$tkyte@ORA10G> select * from t2;
      CNT
---------
        0
```

Итак, у нас имеется процедура, которая, как мы знаем, завершится неудачей. Второй оператор `INSERT` в этом случае всегда провалится. Давайте посмотрим, что случится, если мы запустим эту хранимую процедуру:

```
ops$tkyte@ORA10G> begin
  2 p;
  3 end;
  4 /
Я вставил и обновил 1 строк(у)
Я вставил и обновил 1 строк(у)
begin
*
ERROR at line 1:
ORA-02290: check constraint (OPS$TKYTE.SYS_C009598) violated
ORA-06512: at "OPS$TKYTE.P", line 5
ORA-06512: at line 2
ОШИБКА в строке 1:
ORA-02290: нарушение проверочного ограничения целостности (OPS$TKYTE.SYS_C009598)
ORA-06512: в "OPS$TKYTE.P", строка 5
ORA-06512: в строке 2

ops$tkyte@ORA10G> select * from t;
no rows selected
нет выбранных строк

ops$tkyte@ORA10G> select * from t2;
      CNT
---------
        0
```

Как видите, СУБД Oracle трактует вызов хранимой процедуры как атомарный оператор. Клиент посылает блок кода `BEGIN P; END;`, а Oracle обертывает его в `SAVEPOINT`. Поскольку процедура `P` терпит неудачу, Oracle восстанавливает базу данных в состояние, которое она имела в точке, непосредственно предшествовавшей ее вызову. Теперь, если слегка изменить блок, будет получен совершенно другой результат:

```
ops$tkyte@ORA10G> begin
  2 p;
```

```
  3 exception
  4 when others then null;
  5 end;
  6 /
Я вставил и обновил 1 строк(у)
Я вставил и обновил 1 строк(у)
PL/SQL procedure successfully completed.
Процедура PL/SQL успешно завершена.

ops$tkyte@ORA10G> select * from t;
         X
---------
         1

ops$tkyte@ORA10G> select * from t2;
       CNT
---------
         1
```

Здесь мы запускаем блок кода, игнорирующий все возможные ошибки и отличия в выводе. В то время как первый вызов P не проявляется ни в каких изменениях, этот INSERT выполняется успешно и столбец CNT в T2 соответствующим образом увеличивается.

На заметку! Я считаю ошибочным практически любой код, который содержит обработчик исключений WHEN OTHERS, но не включает вызова RAISE, чтобы передать исключение выше. Он молча игнорирует ошибку и изменяет семантику транзакций. Перехват WHEN OTHERS и трансляция исключения в код возврата в старом стиле изменяет способ ожидаемого поведения базы данных.

СУБД Oracle считает "оператором" любой блок кода, присланный клиентом. Этот оператор завершается успехом за счет самостоятельного перехвата и игнорирования ошибок, поэтому установка If error then rollback... не вступает в действие и Oracle не откатывает работу к SAVEPOINT. Поэтому частичная работа, выполненная P, сохраняется. Причина того, что эта частичная работа предохраняется, состоит в том, что мы имеем автономность уровня оператора внутри P: каждый оператор внутри P является атомарным. P становится клиентом Oracle, когда подтверждает свои два оператора INSERT. Каждый INSERT либо целиком успешен, либо нет. Это подтверждается тем фактом, что мы можем видеть, что триггер T был инициирован дважды и дважды обновил T2, хотя счетчик в T2 отображает только один UPDATE. Второе выполнение INSERT внутри P снабжено неявным SAVEPOINT, обернутым вокруг него.

Отличие между двумя блоками кода довольно-таки тонкое, и вы должны учитывать его в своих приложениях. Добавление обработчика исключений к блоку кода PL/SQL может радикально изменить поведение. Другой способ закодировать это — восстанавливающий атомарность уровня оператора для всего блока PL/SQL, выглядит следующим образом:

```
ops$tkyte@ORA10G> begin
  2      savepoint sp;
  3      p;
  4 exception
```

```
  5      when others then
  6         rollback to sp;
  7 end;
  8 /
Я вставил и обновил 1 строк(у)
Я вставил и обновил 1 строк(у)
PL/SQL procedure successfully completed.
Процедура PL/SQL успешно завершена.

ops$tkyte@ORA10G> select * from t;
no rows selected
нет выбранных строк

ops$tkyte@ORA10G> select * from t2;
      CNT
---------
        0
```

Внимание! Предыдущий код представляет собой пример исключительно порочной практики. Вы никогда не должны перехватывать `WHEN OTHERS`, как и не должны явно кодировать то, что уже обеспечивает Oracle — до тех пор, пока речь идет о семантике транзакции.

Здесь, имитируя то, что Oracle обычно делает для нас с помощью `SAVEPOINT`, мы можем восстановить оригинальное поведение, перехватывая и "игнорируя" ошибку. Я привел этот пример только для иллюстрации; вообще говоря, это исключительно порочная практика.

Атомарность уровня транзакции

И, наконец, существует концепция атомарности уровня транзакции. Общая цель транзакции — набора операторов SQL, выполняемых вместе как единицы работы — состоит в том, чтобы переводить базу данных из одного согласованного состояния в другое.

Для достижения этой цели транзакции также атомарны — полный набор успешной работы, выполненной в транзакции, либо целиком фиксируется и становится постоянным, либо откатывается и отменяется. Так же, как и оператор, транзакция является атомарной единицей работы. По сообщению базы данных об "успехе" после фиксации транзакции вы узнаете о том, что результат работы, выполненная в транзакции, становится постоянным.

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.

На заметку! Считается чрезвычайно плохой практикой выполнять каскадные обновления для модификации первичного ключа. Это нарушает само предназначение первичного ключа. Если вам нужно сделать это однажды, чтобы исправить некорректную информацию — это одно, но если вы постоянно поступаете так в своем приложении, то вам стоит вернуться и заново продумать дизайн, ибо вы выбрали неправильный атрибут в качестве ключа!

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 C, 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.
```

Теперь порядок. В демонстрационных целях я покажу, как явно проверить отложенное ограничение перед фиксацией, чтобы увидеть, согласуются ли модификации с бизнес-правилами (другими словами, проверить, не нарушаются ли ограничения). Будет хорошей идеей сделать это перед фиксацией или передачей управления какой-то другой части программы (которая может не ожидать отложенных ограничений):

```
ops$tkyte@ORA10G> set constraint c_fk immediate;
set constraint c_fk immediate
*
ERROR at line 1:
ORA-02291: integrity constraint (OPS$TKYTE.C_FK) violated - parent key not found
ОШИБКА в строке 1:
ORA-02291: нарушение ограничения целостности (OPS$TKYTE.C_FK) — найдена
родительская запись
```

Попытка завершается неудачей и немедленно возвращает ошибку, как и ожидалось, поскольку мы знали, что ограничение нарушается. Оператор UPDATE таблицы P не откатывается (поскольку это нарушило бы атомарность уровня оператора); он по-прежнему отложен. К тому же обратите внимание, что наша транзакция по-прежнему работает с отложенным ограничением C_FK, поскольку команда SET CONSTRAINT завершилась неудачей. Продолжим каскадным оператором UPDATE для таблицы C:

```
ops$tkyte@ORA10G> update c set fk = 2;
1 row updated.
1 строка обновлена.

ops$tkyte@ORA10G> set constraint c_fk immediate;
Constraint set.
Ограничение установлено.

ops$tkyte@ORA10G> commit;
Commit complete.
Фиксация завершена.
```

Вот так это работает. Следует отметить, что для того, чтобы отложить ограничение, вы должны создать их именно таким образом — удалить и создать заново, чтобы изменить его с неотложенного на отложенное.

Плохие привычки, связанные с транзакциями

Многие разработчики имеют плохие привычки при обращении с транзакциями. Я часто это замечаю у разработчиков, имеющих дело с базами данных, которые "поддерживают", но не "стимулируют" применение транзакций. Например, в Informix (по умолчанию), Sybase и SQL Server, вы должны явно начинать транзакцию словом `BEGIN`; в противном случае каждый индивидуальный оператор сам по себе представляет отдельную транзакцию. В манере, подобной способу, которым Oracle "обертывает" `SAVEPOINT` вокруг дискретных операторов, эти базы данных обертывают каждый оператор парами `BEGIN WORK/COMMIT` или `ROLLBACK`. Это объясняется тем, что в этих базах данных блокировки — ценный ресурс, потому читатели блокируют писателей и наоборот. Пытаясь повысить степень параллелизма, упомянутые базы данных стимулируют вас делать транзакции насколько возможно, краткими — иногда даже за счет целостности данных.

СУБД Oracle исповедует противоположный подход. Транзакции всегда неявны, и нет способа включить "автоматическую фиксацию" ("autocommit"), если только приложение не реализует ее (подробнее об этом читайте в разделе "Использование автоматической фиксации"). В Oracle каждая транзакция должна быть зафиксирована, когда нужно, но никогда раньше. Транзакции могут быть настолько крупными, насколько это необходимо. Такие вещи, как блокировки и тому подобное, не должны рассматриваться в качестве определяющих факторов, диктующих размер транзакции; целостность данных — вот *движущий фактор*, влияющий на размер вашей транзакции. Блокировки не являются ограниченным ресурсом, и нет зависимостей между параллельно работающими читателями и писателями данных.

Это позволяет вам иметь устойчивые транзакции в базе данных. Такие транзакции не обязаны быть краткими — они могут быть настолько длинными, насколько это необходимо (но не больше). Транзакции — не средство достижения согласия между компьютером и его программным обеспечением; они предназначены для того, чтобы защитить ваши данные.

Фиксация в цикле

Сталкиваясь с задачей обновления многих строк, большинство программистов пытаются найти некоторый процедурный способ — выполнить это в цикле, фиксируя каждую из всего этого множества строк. Я слышал два основных аргумента в пользу такого способа.

- Это быстрее и эффективнее — часто фиксировать множество мелких транзакций, чем обрабатывать и фиксировать одну большую транзакцию.
- Нет достаточного пространства для отката.

Оба эти заключения ошибочны. Более того, слишком частая фиксация подвергает вашу базу данных опасности остаться в "неопределенном" состоянии, если процесс обновления даст сбой на полпути. Это требует сложной логики — написать процесс, который будет гладко перезапускаться в случае сбоя. Намного лучше выполнять фиксацию так часто, как этого требует ваш бизнес-процесс, и соответствующим образом настроить размер сегментов отката.

Давайте рассмотрим все это более подробно.

Влияние на производительность

Обычно часто фиксировать получается отнюдь не быстрее — почти всегда быстрее все сделать единственным оператором SQL. Для небольшого примера представим, что у нас есть таблица T с множеством строк, и мы хотим обновить значение столбца для каждой строки в этой таблице. Для демонстрации используются две таблицы — T1 и T2:

```
ops$tkyte@ORA10G> create table t1 as select * from all_objects;
Table created.
Таблица создана.

ops$tkyte@ORA10G> exec dbms_stats.gather_table_stats( user, 'T1' );
PL/SQL procedure successfully completed.
Процедура PL/SQL успешно завершена.

ops$tkyte@ORA10G> create table t2 as select * from t1;
Table created.
Таблица создана.

ops$tkyte@ORA10G> exec dbms_stats.gather_table_stats( user, 'T2' );
PL/SQL procedure successfully completed.
Процедура PL/SQL успешно завершена.
```

Приступая к обновлению, мы можем просто выполнить его единственным оператором UPDATE, как здесь:

```
ops$tkyte@ORA10G> set timing on
ops$tkyte@ORA10G> update t1 set object_name = lower(object_name);
48306 rows updated.
48306 строк обновлено.
Elapsed: 00:00:00.31
Общее время: 00:00:00.31
```

Многие люди по разным причинам предпочитают делать это так, как показано ниже:

```
ops$tkyte@ORA10G> begin
2     for x in ( select rowid rid, object_name, rownum r
3           from t2 )
4     loop
5        update t2
6           set object_name = lower(x.object_name)
7         where rowid = x.rid;
8        if ( mod(x.r,100) = 0 ) then
9           commit;
10    end if;
11 end loop;
12 commit;
13 end;
14 /

PL/SQL procedure successfully completed.
Процедура PL/SQL успешно завершена.
Elapsed: 00:00:05.38
Общее время: 00:00:05.38
```

На этом простом примере видно, что многократная фиксация в цикле в несколько раз замедляет операцию. Если вы можете сделать что-то *единственным* оператором SQL, так и делайте — это почти наверняка будет быстрее. Даже если "оптимизировать" код процедуры, используя групповую обработку обновлений, как показано ниже:

```
ops$tkyte@ORA10G> declare
 2    type ridArray is table of rowid;
 3    type vcArray is table of t2.object_name%type;
 4
 5    l_rids ridArray;
 6    l_names vcArray;
 7
 8    cursor c is select rowid, object_name from t2;
 9 begin
10    open c;
11    loop
12       fetch c bulk collect into l_rids, l_names LIMIT 100;
13       forall i in 1 .. l_rids.count
14          update t2
15             set object_name = lower(l_names(i))
16          where rowid = l_rids(i);
17          commit;
18          exit when c%notfound;
19       end loop;
20       close c;
21 end;
22 /

PL/SQL procedure successfully completed.
Процедура PL/SQL успешно завершена.
Elapsed: 00:00:02.36
Общее время: 00:00:02.36
```

будет действительно намного быстрее, но все же много медленнее, чем могло бы быть. И, кроме того, вы должны заметить, что код становится все более и более сложным. От предельной простоты единственного оператора `UPDATE` к процедурному коду, затем к еще более сложному процедурному коду — мы явно движемся в неверном направлении!

Теперь, просто чтобы поддержать контрапункт этой дискуссии, вспомним главу 7, где мы говорили о концепции согласованной записи и о том, как оператор `UPDATE`, например, может автоматически перезапускаться. В случае, когда процедурный оператор `UPDATE` должен работать с подмножеством записей (имея конструкцию `WHERE`), а другие пользователи модифицируют столбцы, которые этот оператор `UPDATE` использует в своей конструкции `WHERE`, тогда имеет смысл либо использовать серии мелких транзакций вместо одной большой, или же блокировать таблицу перед тем, как выполнять массовое обновление. Целью этого является снижение вероятности перезапусков. Если мы собираемся обновить `UPDATE` значительную часть строк таблицы, это приводит к необходимости применения команды `LOCK TABLE`. Согласно моему опыту, однако, подобного рода массовые обновления или массовые удаления (только этого типа операторы могут быть субъектами перезапуска) выполняются в изоляции. Такие крупные единовременные пакетные

обновления или удаления старых данных обычно не выполняются в период высокой активности. В самом деле, удаление данных вообще не должно быть затронуто этим, поскольку вы обычно работаете с некоторыми полями типа даты, чтобы найти старую информацию, подлежащую удалению, а другие приложения обычно эти даты не модифицируют.

Ошибка "snapshot too old"

Теперь давайте рассмотрим вторую причину, по которой разработчики соблазняются фиксировать обновления в процедурном цикле, и которые происходят из-за безуспешных попыток экономно использовать "ограниченный ресурс" (сегменты отката). Это вопрос конфигурации; вы *должны* обеспечить наличие достаточного пространства для сегментов отмены, чтобы правильно выставить размер ваших транзакций. Фиксация в цикле, помимо того, что работает медленнее, также часто вызывает появление ужасной ошибки ORA-01555. Рассмотрим это более детально.

Как вы знаете после прочтения глав 1 и 7, многоверсионная модель Oracle использует сегменты отмены для реконструирования блоков в том виде, в каком они были на момент запуска вашего оператора или транзакции (в зависимости от режима изоляции). Если необходимая информация отмены больше не существует, вы получаете сообщение об ошибке ORA-01555: snapshot too old ("ORA-1555: устаревший снимок"), и ваш запрос не выполняется. Поэтому, если вы модифицируете таблицу, которую читаете (как в предыдущем примере), то генерируете информацию отмены, необходимую вашему запросу. Оператор UPDATE генерирует информацию отмены, которую ваш запрос, вероятно, использует, чтобы получить согласованное по чтению представление данных, которые ему нужно обновить. Если вы выполняете фиксацию, то позволяете системе повторно задействовать пространство отмены, только что использованное вами. И если она повторно использует это пространство, уничтожая данные отмены, которые впоследствии понадобятся вашему запросу, у вас возникает серьезная проблема. Оператор SELECT завершается неудачей, а UPDATE останавливается на полпути. Вы получаете частично завершенную логическую транзакцию и, вероятно, не имеете никакой возможности перезапустить ее (подробнее об этом ниже).

Давайте рассмотрим сказанное на небольшом примере. В маленькой тестовой базе данных я создал таблицу:

```
ops$tkyte@ORA10G> create table t as select * from all_objects;
Table created.
Таблица создана.

ops$tkyte@ORA10G> create index t_idx on t(object_name);
Index created.
Индекс создан.

ops$tkyte@ORA10G> exec dbms_stats.gather_table_stats( user, 'T', cascade=>true );
PL/SQL procedure successfully completed.
Процедура PL/SQL успешно завершена.
```

Затем я создал очень маленькое табличное пространство отмены и указал системе использовать его. Обратите внимание, что при установке AUTOEXTEND OFF я ограничил размер UNDO в системе до 2 Мбайт или менее:

```
ops$tkyte@ORA10G> create undo tablespace undo_small
  2 datafile size 2m
  3 autoextend off
  4 /
Tablespace created.
Табличное пространство создано.

ops$tkyte@ORA10G> alter system set undo_tablespace = undo_small;
System altered.
Система изменена.
```

Теперь, имея только одно маленькое табличное пространство отмены, я запустил следующий блок кода для выполнения UPDATE:

```
ops$tkyte@ORA10G> begin
 2    for x in ( select /*+ INDEX(t t_idx) */ rowid rid, object_name, rownum r
 3               from t
 4               where object_name > ' ' )
 5    loop
 6       update t
 7          set object_name = lower(x.object_name)
 8        where rowid = x.rid;
 9       if ( mod(x.r,100) = 0 ) then
10          commit;
11       end if;
12    end loop;
13    commit;
14 end;
15 /
begin
*
ERROR at line 1:
ORA-01555: snapshot too old: rollback segment number with name "" too small
ORA-06512: at line 2
ОШИБКА в строке 1:
ORA-01555: устаревший снимок: номер сегмента отката с именем "" очень мал
ORA-06512: в строке 2
```

Возникла ошибка. Хочу отметить, что я добавил подсказку (hint) для использования индекса в запросе и конструкцию WHERE, чтобы гарантировать чтение таблицы в случайном порядке (вместе они заставляют стоимостной оптимизатор читать таблицу, "отсортированную" по ключу индекса). Когда мы обрабатываем таблицу через индекс, то читаем блок для извлечения одной строки, а следующая строка с высокой степенью вероятности должна уже быть в другом блоке. В конечном итоге обрабатываются все строки из блока 1, но не одновременно. Блок 1 может содержать, скажем, данные всех строк, у которых OBJECT_NAME начинается с букв A, M, N, Q и Z. Поэтому мы должны обратиться к этому блоку много раз, поскольку читаем данные, отсортированные по OBJECT_NAME, и можно предположить, что множество значений OBJECT_NAME будут начинаться с букв между A и M. Поскольку фиксация выполняется часто и пространство отката используется повторно, мы со временем повторно навещаем блок, в котором просто невозможно выполнить откат к точке во времени, когда запрос начался, и именно в этот момент получаем ошибку.

Это очень надуманный пример — просто чтобы наглядно продемонстрировать, что может случиться. Мой оператор UPDATE сгенерировал откат. У меня было очень маленькое табличное пространство отката (всего 2 Мбайт). Я многократно заворачивал сегменты отката, поскольку они используются в циклическом режиме. Каждый раз, когда я выполнял фиксацию, то тем самым позволял Oracle перезаписывать сгенерированные данные отката. В конце концов, мне понадобился некоторый кусочек данных, который я сгенерировал, но его не оказалось в сегменте отката, поэтому возникла ошибка ORA-01555.

Вы вправе указать, что в этом случае, если бы я не выполнял фиксацию в строке 10 предыдущего примера, то получил бы следующую ошибку:

```
begin
*
ERROR at line 1:
ORA-30036: unable to extend segment by 8 in undo tablespace 'UNDO_SMALL'
ORA-06512: at line 6
ОШИБКА в строке 1:
ORA-30036: невозможно расширить сегмент на 8 в табличном пространстве
отмены 'UNDO_SMALL'
ORA-06512: в строке 6
```

Ниже описаны главные отличия между этими двумя ошибками.

- Пример с ORA-01555 *оставляет мое обновление в совершенно неопределенном состоянии.* Часть работы сделана, часть — нет.
- Я абсолютно *ничего не могу сделать, чтобы избежать* ORA-01555, если стану выполнять фиксацию в цикле курсора FOR.
- *Ошибки ORA-30036 можно избежать*, выделив соответствующие ресурсы в системе. Эта ошибка устраняется корректным определением размера сегмента отката; первая ошибка — нет. Затем, даже если я не смогу избежать этой ошибки, то по крайней мере обновление откатывается и база данных остается в известном, согласованном состоянии — я не оставляю ее в состоянии частичного обновления.

Главным недостатком здесь является то, что вы не можете "сэкономить" пространство отката за счет частой фиксации — оно вам нужно. Я работал в однопользовательской системе, когда получил ошибку ORA-01555. Достаточно было одного сеанса, чтобы привести к этой ошибке, а в реальной жизни таких ошибок будет множество, если каждый индивидуальный сеанс вызовет свою собственную ошибку ORA-01555. Разработчики и администраторы баз данных должны работать совместно, чтобы адекватно определить размер этих сегментов для задач, решаемых в системе. Здесь не должно быть кратковременных изменений. Путем тщательного анализа вашей системы вы должны выяснить размер наибольшей транзакции и соответствующим образом установить размеры сегментов отката. Динамическое представление производительности V$UNDOSTAT может очень пригодиться для слежения за генерируемыми откатами и продолжительностью самых длительных запросов. Многие люди склонны воспринимать вещи вроде временной области, области отмены и области повторения как "накладные расходы", которые следует сокращать насколько возможно. Это напоминает о проблеме, с которой столкнулась компьютерная индустрия 1 января 2000 года — она была вызвана стремлением сэ-

кономить 2 байта в полях дат. Эти компоненты базы данных не являются накладными расходами, а ключевыми компонентами системы. Их размер должен быть определен должным образом (не слишком большим, но и не слишком маленьким).

Перезапускаемые процессы требуют сложной логики

Наиболее серьезная проблема с подходом "фиксации перед завершением логической транзакции" заключается в том факте, что он часто оставляет вашу базу данных в неопределенном состоянии, если оператор UPDATE прерывается на полпути. Если только вы не позаботитесь об этом заранее, будет очень трудно перезапустить неудавшийся процесс, и позволить ему найти место, в котором он был прерван. Например, предположим, что мы не применяем функцию LOWER() к столбцу, как в предыдущем примере, а некоторую другую функцию столбца, как показано ниже:

```
last_ddl_time = last_ddl_time + 1;
```

Если прервать цикл UPDATE на полпути, как мы сможем перезапустить его? Мы даже не сможем повторно запустить его сначала, потому что в этом случае некоторые данные будут увеличены на 2, а другие — на 1. Если мы опять потерпим неудачу, то при следующем запуске, некоторые увеличатся на 3, другие — на 2, а остальные — на 1. Значит, нужна более сложная логика — какой-то способ "разбиения" данных. Например, можно обработать сначала каждое значение OBJECT_NAME, начинающееся с A, затем — с B и так далее:

```
ops$tkyte@ORA10G> create table to_do
 2 as
 3 select distinct substr( object_name, 1,1 ) first_char
 4 from T
 5 /
Table created.
Таблица создана.

ops$tkyte@ORA10G> begin
 2    for x in ( select * from to_do )
 3    loop
 4       update t set last_ddl_time = last_ddl_time+1
 5        where object_name like x.first_char || '%';
 6
 7       dbms_output.put_line( sql%rowcount || ' rows updated' );
 8       delete from to_do where first_char = x.first_char;
 9
10       commit;
11    end loop;
12 end;
13 /
22257 rows updated
22257 строк обновлено.
1167 rows updated
1167 строк обновлено
135 rows updated
135 строк обновлено
1139 rows updated
1139 строк обновлено
```

```
2993 rows updated
2993 строк обновлено
691 rows updated
691 строк обновлено
...
2810 rows updated
2810 строк обновлено
6 rows updated
6 строк обновлено
10 rows updated
10 строк обновлено
2849 rows updated
2849 строк обновлено
1 rows updated
1 строка обновлена
2 rows updated
2 строк обновлено
7 rows updated
7 строк обновлено
PL/SQL procedure successfully completed.
Процедура PL/SQL успешно завершена.
```

Теперь мы можем перезапустить процесс, если он завершится сбоем, поскольку уже не станем обрабатывать имена объектов, которые были успешно обработаны. Проблема с этим подходом, однако, заключается в том, что если только у нас нет атрибута, позволяющего равномерно распределить данные, то мы столкнемся с очень широким распределением строк. Первому `UPDATE` придется выполнить намного больше работы, чем всем прочим вместе взятым. Вдобавок если другие сеансы обращаются к этой же таблице и модифицируют данные, они также могут обновить поле `object_name`. Предположим, что некоторый другой сеанс обновляет объект по имени Z, переименовывая его в A, *после* того, как мы уже обработали все объекты на A. В этом случае мы потеряем эту запись. Более того, это очень неэффективный процесс по сравнению с `UPDATE T SET LAST_DDL_TIME = LAST_DDL_TIME+1`. Возможно, мы воспользуемся индексом для чтения каждой строки таблицы или выполним полное сканирование `n` раз — и то, и другое нежелательно. О таком подходе можно сказать очень много плохого.

И поэтому лучший подход — первый, который описан в начале главы 1: делать это просто. Если это можно сделать на SQL — делайте это на SQL. Что невозможно сделать на SQL, делайте на PL/SQL. Делайте это, используя минимально возможный объем кода. Выделяйте достаточные ресурсы. Всегда думайте о том, что может произойти в случае ошибки. Мне очень часто приходилось видеть людей, кодирующих циклы обновления, которые великолепно работали с тестовыми данными, но терпели крах при обработке реальных данных. Они попадали впросак, поскольку не имели понятия, где именно прервалась обработка. Намного проще выставить правильно размер сегмента отката, нежели писать перезапускаемую программу. Если у вас есть действительно большие таблицы, которые нужно обновлять, то вы должны использовать секционирование (подробнее о нем рассказывается в главе 10), которое позволит вам обновить каждую секцию индивидуально. Тогда вы можете даже применять параллельный DML для выполнения обновлений.

Использование автоматической фиксации

Мое последнее слово о плохих привычках, связанных с транзакциями, касается той, что происходит от использования популярных программных интерфейсов — ODBS и JDBC. Эти API выполняют автоматическую фиксацию ("autocommit") по умолчанию. Рассмотрим следующие операторы, которые переводят сумму $1000 с расчетного счета на сберегательный счет:

```
update accounts set balance = balance - 1000 where account_id = 123;
update accounts set balance = balance + 1000 where account_id = 456;
```

Если ваша программа использует JDBC, когда вы посылаете эти запросы, то JDBC (молча) выполняет фиксацию после *каждого* оператора `UPDATE`. Представьте последствия этого, если система даст сбой после первого `UPDATE` и перед вторым. Вы потеряете $1000!

Я могу понять, почему ODBC так поступает. ODBC проектировали разработчики SQL Server, а эта база данных требует, чтобы вы применяли очень короткие транзакции из-за используемой там модели параллелизма (запись блокирует чтение, чтение блокирует запись, а блокировки — ограниченный ресурс). Чего я не могу понять — так это почему подобное попало в JDBC — API, претендующий на поддержку систем уровня предприятия. Поэтому я уверен в том, что первой строкой кода после открытия соединения в JDBC всегда должна быть следующая:

```
connection conn = DriverManager.getConnection
("jdbc:oracle:oci:@database","scott","tiger");
conn.setAutoCommit (false);
```

Это вернет управление транзакциями вам, то есть разработчику, которому оно и должно принадлежать. Затем вы сможете безопасно кодировать транзакцию перевода денег с одного счета на другой и фиксировать ее после того, как оба оператора успешно завершатся. Недостаток знаний о вашем API может быть смертельным в этом случае. Я видел не одного разработчика, который, не зная об этом "средстве" автоматической фиксации, имел серьезнейшие проблемы, когда его приложение сталкивалось с ошибкой.

Распределенные транзакции

Одним из действительно замечательных свойств СУБД Oracle является ее способность прозрачно обрабатывать распределенные транзакции. Я могу обновить данные во множестве разных баз данных за одну транзакцию. Когда я выполняю ее фиксацию, то либо фиксируются все обновления во всех экземплярах, либо не фиксируется ни одно из них (все они откатываются). Мне не нужен для этого никакой дополнительный код; я просто пишу `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. Однако цель настоящего раздела не в том, чтобы раскрыть синтаксис команд связи баз данных (он полностью документирован), а скорее просто в том, чтобы уведомить вас о его существовании. 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, я также могу 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;
```

Вот и все. 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. Если вы помните дискуссию об архитектуре из главы 5, там было сказано,

что 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 (на мой взгляд, они выглядят оправданными в любом случае). 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 10*g* — 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, я хотел бы подчеркнуть, что это — весьма мощный, а потому опасный инструмент, если применять его неправильно. Реальная потребность в автономных транзакциях возникает очень редко. Я с подозрением отношусь к любому коду, который использует их — к такому коду следует внимательно присмотреться. Слишком легко непреднамеренно нарушить логическую целостность данных в системе с автономными транзакциями. В следующем разделе мы обсудим, когда они могут безопасно применяться, после того, как посмотрим, как они работают.

Как работают автономные транзакции

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 — "нормальная" 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.

Когда использовать автономные транзакции

СУБД Oracle внутри себя достаточно давно поддерживает автономные транзакции. Мы видим их в форме рекурсивного SQL. Например, рекурсивная транзакция может выполняться при выборе значения из последовательности, чтобы немедленно увеличить значение счетчика последовательности в таблице `SYS.SEQ$`. Обновление таблицы `SYS.SEQ$` для поддержки вашей последовательности немедленно фиксируется и становится видимым другим транзакциям, хотя ваша транзакция еще и не зафиксирована. Вдобавок, если вы откатите свою транзакцию, то увеличение значения счетчика последовательности останется в силе; оно не будет откачено вместе с вашей транзакцией, поскольку уже было зафиксировано. Управление пространством, аудит и другие внутренние операции выполняются в аналогичной рекурсивной манере.

Теперь это средство открыто для применения всем. Однако я обнаружил, что оправданное применение автономных транзакций в реальном мире *очень ограничено*. Временами я встречаю их в качестве обходного пути для решения таких проблем, как ограничения мутирующей таблицы в триггере. Однако это почти всегда ведет к проблемам с целостностью данных, поскольку причиной мутирующей

таблицы является попытка читать таблицу в то время, когда инициирован ее триггер. Используя автономную транзакцию, вы можете запросить таблицу, но при этом не сможете увидеть изменения (для чего в первую очередь и предназначено ограничение мутирующей таблицы; таблица находится в процессе модификации, поэтому результаты запроса должны быть несогласованными). Любые решения, принятые на основе запроса из триггера, сомнительны — в этот момент времени вы читаете "старые" данные.

Потенциально корректное применение автономной транзакции — в специализированном аудите, но я подчеркиваю слова "потенциально корректное". Существуют более эффективные пути аудита информации базы данных, нежели посредством написания специального триггера. Например, вы можете использовать пакет `DBMS_FGA` или саму команду `AUDIT`.

Часто задаваемый разработчиками приложений вопрос звучит так: "Как я могу выполнять аудит каждой попытки модификации секретной информации и записывать значения, которые они пытаются модифицировать?" Они хотят не только *предотвращать* попытки модификации, но также сохранять записи о таких попытках. До появления автономных транзакций многие разработчики пытались (и безуспешно) делать это с помощью стандартных триггеров без автономных транзакций. Триггер должен обнаруживать `UPDATE`, и, увидев, что пользователь модифицирует данные, которых он модифицировать не должен, создавать запись аудита и отказывать `UPDATE`. К сожалению, когда триггер отвергает `UPDATE`, он также выполняет откат записи аудита — это типичная ситуация "все или ничего". С автономными транзакциями стало возможным безопасно выполнять аудит попыток выполнения запретных операций, одновременно откатывая эти операции. В процессе мы можем информировать конечного пользователя о том, что он пытается модифицировать данные, которые не имеет права модифицировать, а также о том, что его попытка зафиксирована в записи аудита.

Интересно отметить, что родная команда Oracle `AUDIT` уже в течение многих лет предоставляет возможность фиксации безуспешных попыток модификации информации, используя автономные транзакции. Предоставление этого средства в распоряжение разработчиков Oracle позволяет нам выполнять собственный, более гибкий специализированный аудит.

Рассмотрим небольшой пример. Давайте добавим триггер автономной транзакции к таблице, которая фиксирует след аудита, детализируя, *кто* пытался обновить таблицу, и *когда* это произошло, наряду с информативным сообщением о том, какие именно данные этот пользователь пытался модифицировать. Логика такого триггера должна предотвращать любые попытки обновления записи сотрудника, который (прямо или опосредованно) не подчиняется вам.

Во-первых, создадим копию таблицы `EMP` из схемы `SCOTT`, чтобы использовать в качестве таблицы примеров:

```
ops$tkyte@ORA10G> create table emp
  2 as
  3 select * from scott.emp;
Table created.
Таблица создана.

ops$tkyte@ORA10G> grant all on emp to scott;
Grant succeeded.
Полномочия унаследованы.
```

Мы также создадим таблицу AUDIT_TAB, в которой будем размещать информацию аудита. Обратите внимание, что мы используем атрибут DEFAULT в столбцах, которые должны хранить имя текущего зарегистрированного пользователя и текущее значение даты/времени в наш "след" аудита.

```
ops$tkyte@ORA10G> create table audit_tab
  2 ( username varchar2(30) default user,
  3   timestamp date default sysdate,
  4   msg varchar2(4000)
  5 )
  6 /
Table created.
Таблица создана.
```

Далее создадим триггер EMP_AUDIT для выполнения аудита действия UPDATE над таблицей EMP:

```
ops$tkyte@ORA10G> create or replace trigger EMP_AUDIT
  2 before update on emp
  3 for each row
  4 declare
  5      pragma autonomous_transaction;
  6      l_cnt number;
  7 begin
  8
  9      select count(*) into l_cnt
 10        from dual
 11       where EXISTS ( select null
 12                      from emp
 13                     where empno = :new.empno
 14                     start with mgr = ( select empno
 15                                          from emp
 16                                         where ename = USER )
 17                connect by prior empno = mgr );
 18      if ( l_cnt = 0 )
 19      then
 20         insert into audit_tab ( msg )
 21         values ( 'Attempt to update ' || :new.empno );
 22         commit;
 23
 24         raise_application_error( -20001, 'Access Denied' );
 25      end if;
 26 end;
 27 /
Trigger created.
Триггер создан.
```

Обратите внимание на применение запроса с CONNECT BY. Это позволяет нам развернуть всю иерархию, начиная с текущего пользователя, чтобы проверить, что запись, попытка обновления которой предпринимается, относится к кому-то, кто подчиняется нам на некотором уровне.

Ниже перечислены главные моменты, которые нужно отметить относительно этого триггера.

- PRAGMA AUTONOMOUS_TRANSACTION применяется к определению триггера. Весь триггер представляет собой "автономную транзакцию", и потому независим от родительской транзакции, пытающейся выполнить обновление.
- Триггер пытается выполнять чтение таблицы, которую он защищает, а именно — таблицу EMP. Само по себе это может привести к ошибке "мутирующей" таблицы во время выполнения, но это не касается автономной транзакции. Автономная транзакция обходит эту проблему — она позволяет читать таблицу, но с условием, что мы не увидим тех изменений, которые проводятся в таблице вышестоящей транзакцией. В таком случае следует проявлять исключительную осторожность. Подобная логика должна быть тщательно продумана. Что если текущая транзакция пытается обновить саму иерархию сотрудников? Мы не увидим этих изменений в триггере, и это следует принимать во внимание, проверяя корректность кода триггера.
- Этот триггер выполняет фиксацию. Раньше такое было невозможно — триггеры никогда не фиксировали работу. Данный триггер не фиксирует ту работу, которая была причиной его вызова; вместо этого он фиксирует только ту работу, которую выполняет он сам (запись аудита).

Итак, мы настроили таблицу EMP, имеющую замечательную иерархическую структуру (рекурсивное отношение EMPNO — MGR). Также у нас есть таблица AUDIT_TABLE, в которую мы хотим вносить записи о безуспешных попытках модифицировать информацию. У нас имеется триггер, обеспечивающий выполнение нашего правила — что только наш руководитель или руководитель нашего руководителя (и так далее) может модифицировать нашу запись.

Давайте посмотрим, как это работает, попытавшись обновить запись в таблице EMP:

```
ops$tkyte@ORA10G> update emp set sal = sal*10;
update emp set sal = sal*10
       *
ERROR at line 1:
ORA-20001: Access Denied
ORA-06512: at "OPS$TKYTE.EMP_AUDIT", line 21
ORA-04088: error during execution of trigger 'OPS$TKYTE.EMP_AUDIT'
ОШИБКА в строке 1:
ORA-20001: Доступ запрещен
ORA-06512: в "OPS$TKYTE.EMP_AUDIT", строка 21
ORA-04088: ошибка во время выполнения триггера 'OPS$TKYTE.EMP_AUDIT'

ops$tkyte@ORA10G> select * from audit_tab;

USERNAME   TIMESTAMP  MSG
---------  ---------  ----------------------------------------
OPS$TKYTE  27-APR-05  Attempt to update 7369
```

Вызов триггера предотвращает операцию UPDATE, в то же время создавая постоянную запись о такой попытке (обратите внимание, как используется ключевое слово DEFAULT в операторе CREATE TABLE таблицы AUDIT_TAB, чтобы автоматически вносить значения USER и SYSDATE). Далее давайте зарегистрируемся как пользователь, который может выполнить UPDATE, и попытаемся сделать кое-что:

```
ops$tkyte@ORA10G> connect scott/tiger
Connected.
Подключено.

scott@ORA10G> set echo on
scott@ORA10G> update ops$tkyte.emp set sal = sal*1.05 where ename = 'ADAMS';
1 row updated.
1 строка обновлена.

scott@ORA10G> update ops$tkyte.emp set sal = sal*1.05 where ename = 'SCOTT';
update ops$tkyte.emp set sal = sal*1.05 where ename = 'SCOTT'
              *
ERROR at line 1:
ORA-20001: Access Denied
ORA-06512: at "OPS$TKYTE.EMP_AUDIT", line 21
ORA-04088: error during execution of trigger 'OPS$TKYTE.EMP_AUDIT'
ОШИБКА в строке 1:
ORA-20001: Доступ запрещен
ORA-06512: в "OPS$TKYTE.EMP_AUDIT", строка 21
ORA-04088: ошибка во время выполнения триггера 'OPS$TKYTE.EMP_AUDIT'
```

В инсталляции по умолчанию демонстрационной таблицы EMP сотрудник ADAMS подчиняется SCOTT, поэтому первый оператор UPDATE выполнился успешно. Второй UPDATE, где SCOTT пытается дать самому себе прибавку, терпит провал, поскольку SCOTT не подчиняется SCOTT. Зарегистрируемся в схеме, содержащей таблицу AUDIT_TAB, и увидим следующее:

```
scott@ORA10G> connect /
Connected.
Подключено.

ops$tkyte@ORA10G> set echo on
ops$tkyte@ORA10G> select * from audit_tab;

USERNAME   TIMESTAMP  MSG
---------  ---------  ----------------------------------------
OPS$TKYTE  27-APR-05  Attempt to update 7369
SCOTT      27-APR-05  Attempt to update 7788
```

Как видим, попытка SCOTT выполнить оператор UPDATE была зафиксирована.

Резюме

В данной главе мы рассмотрели множество аспектов управления транзакциями в Oracle. 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.

Мы рассмотрели некоторые сложные взаимодействия между ограничениями целостности (уникальные ключи, проверочные ограничения и тому подобные) и транзакциями в Oracle. Мы обсудили, как обычно Oracle обрабатывает ограничения целостности немедленно после выполнения операторов, а затем — как можно отложить верификацию ограничений до конца транзакции, когда это необходимо. Это средство является ключевым в реализации сложных многотабличных обновлений, когда модифицируемые таблицы зависят друг от друга — примером служило каскадное обновление.

Мы проанализировали некоторые вредные привычки разработчиков, связанные с транзакциями, которые происходят из опыта работы с базами данных, "поддерживающими", а не "стимулирующими" к применению транзакций. Мы описали кардинальное правило транзакций: они должны быть насколько возможно краткими, но не короче, чем это необходимо. *Целостность данных управляет размером транзакций* — это ключевая концепция, которую вы должны вынести из этой главы. Единственный фактор, влияющий на размер ваших транзакций — это бизнес-правила, которым подчиняется ваша система. Нет пространства отката, нет блокировок — вот каковы бизнес-правила.

Мы раскрыли тему распределенных транзакций и их отличие от одиночных транзакций баз данных. Мы исследовали ограничения, накладываемые на распределенные транзакции, и объяснили причины их существования. Прежде чем вы построите распределенную систему, вам следует четко понимать эти ограничения. То, что работает в отдельном экземпляре, может не работать в распределенной базе данных.

Эта глава завершилась рассмотрением автономных транзакций с описанием того, что они собой представляют, и, что более важно — когда их стоит использовать, а когда — нет. Я хотел бы еще раз подчеркнуть, что оправданное применение автономных транзакций в реальном мире случается исключительно редко. Если вы обнаружите, что используете их постоянно, вам стоит серьезно задуматься над причинами этого.