Глава 9

Транзакции

Детальное понимание механизмов выполнения транзакций придет с опытом. В этой главе мы дадим самое первое представление об этом важном и мощном инструменте, которым обладают все серьезные СУБД, включая PostgreSQL.

9.1. Общая информация

Транзакция — это совокупность операций над базой данных, которые вместе образуют логически целостную процедуру, и могут быть либо выполнены все вместе, либо не будет выполнена ни одна из них. В простейшем случае транзакция состоит из одной операции.

Транзакции являются одним из средств обеспечения согласованности (непротиворечивости) базы данных, наряду с ограничениями целостности (constraints), накладываемыми на таблицы. Транзакция переводит базу данных из одного согласованного состояния в другое согласованное состояние.

В качестве примера транзакции в базе данных «Авиаперевозки» можно привести процедуру бронирования билета. Она будет включать операции `INSERT`, выполняемые над таблицами «Бронирования» (`bookings`), «Билеты» (`tickets`) и «Перелеты» (`ticket_flights`). В результате выполнения этой транзакции должно обеспечиваться следующее соотношение: значение атрибута `total_amount` в строке таблицы `bookings` должно быть равно сумме значений атрибута `amount` в строках таблицы `ticket_flights`, связанных с этой строкой таблицы `bookings`. Если операции данной транзакции будут выполнены частично, тогда может оказаться, например, что общая сумма бронирования будет не равна сумме стоимостей перелетов, включенных в это бронирование. Очевидно, что это несогласованное состояние базы данных.

Транзакция может иметь два исхода: первый — изменения данных, произведенные в ходе ее выполнения, успешно зафиксированы в базе данных, а второй исход таков — транзакция отменяется, и отменяются все изменения, выполненные в ее рамках. Отмена транзакции называется откатом (rollback).

Сложные информационные системы, как правило, предполагают одновременную работу многих пользователей с базой данных, поэтому современные СУБД предлагают специальные механизмы для организации параллельного, т. е. одновременного, выполнения транзакций. Реализованы такие механизмы и в PostgreSQL.

Реализация транзакций в СУБД PostgreSQL основана на многоверсионной модели (Multiversion Concurrency Control, MVCC). Эта модель предполагает, что каждый SQL-оператор видит так называемый *снимок* данных (snapshot), т. е. то согласованное состояние (версию) базы данных, которое она имела на определенный момент времени. При этом параллельно исполняемые транзакции, даже вносящие изменения в базу данных, не нарушают согласованности данных этого снимка. Такой результат в PostgreSQL достигается за счет того, что когда параллельные транзакции изменяют одни и те же строки таблиц, тогда создаются отдельные *версии* этих строк, доступные соответствующим транзакциям. Это позволяет ускорить работу с базой данных, однако требует больше дискового пространства и оперативной памяти. И еще одно важное следствие применения MVCC — операции чтения никогда не блокируются операциями записи, а операции записи никогда не блокируются операциями чтения.

Согласно теории баз данных транзакции должны обладать следующими свойствами:

1. Атомарность (atomicity). Это свойство означает, что либо транзакция будет зафиксирована в базе данных полностью, т. е. будут зафиксированы результаты выполнения всех ее операций, либо не будет зафиксирована ни одна операция транзакции.
2. Согласованность (consistency). Это свойство предписывает, чтобы в результате успешного выполнения транзакции база данных была переведена из одного согласованного состояния в другое согласованное состояние.
3. Изолированность (isolation). Во время выполнения транзакции другие транзакции должны оказывать по возможности минимальное влияние на нее.
4. Долговечность (durability). После успешной фиксации транзакции пользователь должен быть уверен, что данные надежно сохранены в базе данных и впоследствии могут быть извлечены из нее, независимо от последующих возможных сбоев в работе системы.

Для обозначения всех этих четырех свойств используется аббревиатура ACID.

При параллельном выполнении транзакций возможны следующие феномены:

1. Потерянное обновление (lost update). Когда разные транзакции одновременно изменяют одни и те же данные, то после фиксации изменений может оказаться,

что одна транзакция перезаписала данные, обновленные и зафиксированные другой транзакцией.

2. «Грязное» чтение (dirty read). Транзакция читает данные, измененные параллельной транзакцией, которая еще не завершилась. Если эта параллельная транзакция в итоге будет отменена, тогда окажется, что первая транзакция прочитала данные, которых нет в системе.

3. Неповторяющееся чтение (non-repeatable read). При повторном чтении тех же самых данных в рамках одной транзакции оказывается, что другая транзакция успела изменить и зафиксировать эти данные. В результате тот же самый запрос выдает другой результат.

4. Фантомное чтение (phantom read). Транзакция повторно выбирает множество строк в соответствии с одним и тем же критерием. В интервале времени между выполнением этих выборок другая транзакция добавляет новые строки и успешно фиксирует изменения. В результате при выполнении повторной выборки в первой транзакции может быть получено другое множество строк.

5. Аномалия сериализации (serialization anomaly). Результат успешной фиксации группы транзакций, выполняющихся параллельно, не совпадает с результатом ни одного из возможных вариантов упорядочения этих транзакций, если бы они выполнялись последовательно.

Перечисленные феномены, а также ситуации, в которых они имеют место, будут рассмотрены подробно и проиллюстрированы примерами.

Поясним кратко, в чем состоит смысл концепции сериализации. Для двух транзакций, скажем, A и B, возможны только два варианта упорядочения при их последовательном выполнении: сначала A, затем B или сначала B, затем A. Причем результаты реализации двух вариантов могут в общем случае не совпадать. Например, при выполнении двух банковских операций — внесения некоторой суммы денег на какой-то счет и начисления процентов по этому счету — важен порядок выполнения операций. Если первой операцией будет увеличение суммы на счете, а второй — начисление процентов, тогда итоговая сумма будет больше, чем при противоположном порядке выполнения этих операций. Если описанные операции выполняются в рамках двух различных транзакций, то оказываются возможными различные итоговые результаты, зависящие от порядка их выполнения.

Сериализация двух транзакций при их *параллельном* выполнении означает, что полученный результат будет соответствовать *одному* из двух возможных вариантов упорядочения транзакций при их последовательном выполнении. При этом нельзя сказать точно, какой из вариантов будет реализован.

Если распространить эти рассуждения на случай, когда параллельно выполняется более двух транзакций, тогда результат их параллельного выполнения также должен быть таким, каким он был бы в случае выбора *некоторого варианта* упорядочения транзакций, если бы они выполнялись последовательно, одна за другой. Конечно, чем больше транзакций, тем больше вариантов их упорядочения. Концепция сериализации не предписывает выбора какого-то определенного варианта. Речь идет лишь об одном из них.

В том случае, если СУБД не сможет гарантировать успешную сериализацию группы параллельных транзакций, тогда некоторые из них могут быть завершены с ошибкой. Эти транзакции придется выполнить повторно.

Для конкретизации степени независимости параллельных транзакций вводится понятие уровня *изоляции транзакций*. Каждый уровень характеризуется перечнем тех феноменов, которые на данном уровне не допускаются.

Всего в стандарте SQL предусмотрено четыре уровня. Каждый более высокий уровень включает в себя все возможности предыдущего.

1. Read Uncommitted. Это самый низкий уровень изоляции. Согласно стандарту SQL на этом уровне допускается чтение «грязных» (незафиксированных) данных. Однако в PostgreSQL требования, предъявляемые к этому уровню, более строгие, чем в стандарте: чтение «грязных» данных на этом уровне не допускается.

2. Read Committed. Не допускается чтение «грязных» (незафиксированных) данных. Таким образом, в PostgreSQL уровень Read Uncommitted совпадает с уровнем Read Committed. Транзакция может видеть только те незафиксированные изменения данных, которые произведены в ходе выполнения ее самой.

3. Repeatable Read. Не допускается чтение «грязных» (незафиксированных) данных и неповторяющееся чтение. В PostgreSQL на этом уровне не допускается также фантомное чтение. Таким образом, реализация этого уровня является более строгой, чем того требует стандарт SQL. Это не противоречит стандарту.

4. Serializable. Не допускается ни один из феноменов, перечисленных выше, в том числе и аномалии сериализации.

Конкретный уровень изоляции обеспечивает сама СУБД с помощью своих внутренних механизмов. Его достаточно указать в команде при старте транзакции. Однако программист может дополнительно использовать некоторые операторы и приемы программирования, например, устанавливать блокировки на уровне отдельных строк или всей таблицы. Это будет показано в конце главы.

По умолчанию PostgreSQL использует уровень изоляции Read Committed.

```
SHOW default_transaction_isolation;
```

```
 default_transaction_isolation
-------------------------------
 read committed
(1 строка)
```

9.2. Уровень изоляции Read Uncommitted

Давайте начнем рассмотрение с уровня изоляции Read Uncommitted. Проверим, видит ли транзакция те изменения данных, которые были произведены в другой транзакции, но еще не были зафиксированы, т. е. «грязные» данные.

Для проведения экспериментов воспользуемся таблицей «Самолеты» (`aircrafts`). Но можно создать копию этой таблицы, чтобы при удалении строк из нее не удалялись строки из таблицы «Места» (`seats`), связанные по внешнему ключу со строками из таблицы `aircrafts`.

```
CREATE TABLE aircrafts_tmp
  AS SELECT * FROM aircrafts;
```

```
SELECT 9
```

Для организации выполнения параллельных транзакций с использованием утилиты psql будем запускать ее на двух терминалах.

Итак, для изучения уровня изоляции Read Uncommitted проделаем следующие эксперименты.

На первом терминале выполним следующие команды:

```
BEGIN;
```

```
BEGIN
```

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
```

```
SET
```

```
SHOW transaction_isolation;

 transaction_isolation
-----------------------
 read uncommitted
(1 строка)

UPDATE aircrafts_tmp
  SET range = range + 100
  WHERE aircraft_code = 'SU9';

UPDATE 1

SELECT *
  FROM aircrafts_tmp
  WHERE aircraft_code = 'SU9';

 aircraft_code |        model         | range
---------------+----------------------+-------
 SU9           | Sukhoi SuperJet-100 |  3100
(1 строка)
```

Начнем транзакцию на втором терминале (все, что происходит на втором терминале, показано на сером фоне):

```
BEGIN;

BEGIN
```

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

SET
```

```
SELECT *
  FROM aircrafts_tmp
  WHERE aircraft_code = 'SU9';

 aircraft_code |        model         | range
---------------+----------------------+-------
 SU9           | Sukhoi SuperJet-100 |  3000
(1 строка)
```

Таким образом, вторая транзакция не видит изменение значения атрибута `range`, произведенное в первой — незафиксированной — транзакции. Это объясняется тем, что в PostgreSQL реализация уровня изоляции Read Uncommitted более строгая, чем

того требует стандарт языка SQL. Фактически этот уровень тождественен уровню изоляции Read Committed. Поэтому будем считать эксперимент, проведенный для уровня изоляции Read Uncommitted, выполненным и для уровня Read Committed.

Давайте не будем фиксировать произведенное изменение в базе данных, а воспользуемся командой ROLLBACK для отмены транзакции, т. е. для ее отката.

На первом терминале:

```
ROLLBACK;

ROLLBACK
```

На втором терминале сделаем так же:

```
ROLLBACK;

ROLLBACK
```

9.3. Уровень изоляции Read Committed

Теперь обратимся к уровню изоляции Read Committed. Именно этот уровень установлен в PostgreSQL по умолчанию. Мы уже показали, что на этом уровне изоляции не допускается чтение незафиксированных данных. А сейчас покажем, что на этом уровне изоляции также гарантируется отсутствие потерянных обновлений, но возможно неповторяющееся чтение данных.

Опять будем работать на двух терминалах. В первой транзакции увеличим значение атрибута `range` для самолета Sukhoi SuperJet-100 на 100 км, а во второй транзакции — на 200 км. Проверим, какое из этих двух изменений будет записано в базу данных.

На первом терминале выполним следующие команды:

```
BEGIN ISOLATION LEVEL READ COMMITTED;

BEGIN

SHOW transaction_isolation;

 transaction_isolation
-----------------------
 read committed
(1 строка)
```

```
UPDATE aircrafts_tmp
  SET range = range + 100
  WHERE aircraft_code = 'SU9';
```

```
UPDATE 1
```

```
SELECT *
  FROM aircrafts_tmp
  WHERE aircraft_code = 'SU9';
```

```
 aircraft_code |        model         | range
---------------+----------------------+-------
 SU9           | Sukhoi SuperJet-100 |  3100
(1 строка)
```

Мы видим, что в первой транзакции значение атрибута `range` было успешно изменено, хотя пока и не зафиксировано. Но транзакция видит изменения, выполненные в ней самой.

Обратите внимание, что вместо использования команды `SET TRANSACTION` мы просто включили указание уровня изоляции непосредственно в команду `BEGIN`. Эти два подхода равносильны. Конечно, когда речь идет об использовании уровня изоляции Read Committed, принимаемого по умолчанию, можно вообще ограничиться только командой `BEGIN` без дополнительных ключевых слов.

На втором терминале так и сделаем. Во второй транзакции попытаемся обновить эту же строку таблицы `airctafts_tmp`, но для того, чтобы впоследствии разобраться, какое из изменений прошло успешно и было зафиксировано, добавим к значению атрибута `range` не 100, а 200.

```
BEGIN;

BEGIN
```

```
UPDATE aircrafts_tmp
  SET range = range + 200
  WHERE aircraft_code = 'SU9';
```

И вот мы видим, что команда `UPDATE` во второй транзакции не завершилась, а перешла в состояние ожидания. Это ожидание продлится до тех пор, пока не завершится первая транзакция. Дело в том, что команда `UPDATE` в первой транзакции заблокировала строку в таблице `airctafts_tmp`, и эта блокировка будет снята только при завершении транзакции либо с фиксацией изменений с помощью команды `COMMIT`, либо с отменой изменений по команде `ROLLBACK`.

Давайте завершим первую транзакцию с фиксацией изменений:

```
COMMIT;

COMMIT
```

Перейдя на второй терминал, мы увидим, что команда UPDATE завершилась:

```
UPDATE 1
```

Теперь на втором терминале, не завершая транзакцию, посмотрим, что стало с нашей строкой в таблице aircrafts_tmp:

```
SELECT *
  FROM aircrafts_tmp
  WHERE aircraft_code = 'SU9';

 aircraft_code |        model         | range
---------------+----------------------+-------
 SU9           | Sukhoi SuperJet-100 |  3300
(1 строка)
```

Как видно, были произведены оба изменения. Команда UPDATE во второй транзакции, получив возможность заблокировать строку после завершения первой транзакции и снятия ею блокировки с этой строки, *перечитывает* строку таблицы и потому обновляет строку, уже обновленную в только что зафиксированной транзакции. Таким образом, эффекта потерянных обновлений не возникает.

Завершим транзакцию на втором терминале, но вместо команды COMMIT воспользуемся эквивалентной командой END, которая является расширением PostgreSQL:

```
END;

COMMIT
```

Если вы самостоятельно проведете только что выполненный эксперимент, выбрав уровень изоляции Read Uncommitted, то увидите, что и на этом — самом низком — уровне изоляции эффекта потерянных обновлений также не возникает.

Для иллюстрации эффекта неповторяющегося чтения данных проведем совсем простой эксперимент также на двух терминалах. На первом терминале:

```
BEGIN;

BEGIN
```

```
SELECT * FROM aircrafts_tmp;
```

```
 aircraft_code |        model         | range
---------------+----------------------+-------
 773           | Boeing 777-300       | 11100
 763           | Boeing 767-300       |  7900
 320           | Airbus A320-200      |  5700
 321           | Airbus A321-200      |  5600
 319           | Airbus A319-100      |  6700
 733           | Boeing 737-300       |  4200
 CN1           | Cessna 208 Caravan   |  1200
 CR2           | Bombardier CRJ-200   |  2700
 SU9           | Sukhoi SuperJet-100  |  3300
(9 строк)
```

На втором терминале:

```
BEGIN;

BEGIN
```

```
DELETE FROM aircrafts_tmp
  WHERE model ~ '^Boe';

DELETE 3
```

```
SELECT * FROM aircrafts_tmp;

 aircraft_code |        model         | range
---------------+----------------------+-------
 320           | Airbus A320-200      |  5700
 321           | Airbus A321-200      |  5600
 319           | Airbus A319-100      |  6700
 CN1           | Cessna 208 Caravan   |  1200
 CR2           | Bombardier CRJ-200   |  2700
 SU9           | Sukhoi SuperJet-100  |  3300
(6 строк)
```

Сразу завершим вторую транзакцию:

```
END;

COMMIT
```

Повторим выборку в первой транзакции:

```
SELECT * FROM aircrafts_tmp;
```

```
 aircraft_code |         model       | range
---------------+---------------------+-------
 320           | Airbus A320-200     |  5700
 321           | Airbus A321-200     |  5600
 319           | Airbus A319-100     |  6700
 CN1           | Cessna 208 Caravan  |  1200
 CR2           | Bombardier CRJ-200  |  2700
 SU9           | Sukhoi SuperJet-100 |  3300
(6 строк)
```

Видим, что теперь получен другой результат, т. к. вторая транзакция завершилась в момент времени между двумя запросами. Таким образом, налицо эффект неповторяющегося чтения данных, который является допустимым на уровне изоляции Read Committed.

Завершим и первую транзакцию:

```
END;
```

```
COMMIT
```

9.4. Уровень изоляции Repeatable Read

Третий уровень изоляции — Repeatable Read. Само его название говорит о том, что он не допускает феномен неповторяющегося чтения данных. А в PostgreSQL на этом уровне не допускается и чтение фантомных строк.

Приложения, использующие этот уровень изоляции, должны быть готовы к тому, что придется выполнять транзакции повторно. Это объясняется тем, что транзакция, использующая этот уровень изоляции, создает снимок данных не перед выполнением каждого запроса, а только однократно, перед выполнением *первого запроса* транзакции. Поэтому транзакции с этим уровнем изоляции не могут изменять строки, которые были изменены другими завершившимися транзакциями уже после создания снимка. Вследствие этого PostgreSQL не позволит зафиксировать транзакцию, которая попытается изменить уже измененную строку.

Важно помнить, что повторный запуск может потребоваться только для транзакций, которые вносят изменения в данные. Для транзакций, которые только читают данные, повторный запуск никогда не требуется.

На первом терминале:

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;

BEGIN
```

Сначала посмотрим содержимое таблицы:

```
SELECT * FROM aircrafts_tmp;
```

Обратите внимание, что после уже проведенных экспериментов в таблице осталось меньше строк, чем было вначале.

```
 aircraft_code |        model         | range
---------------+----------------------+-------
 320           | Airbus A320-200      |  5700
 321           | Airbus A321-200      |  5600
 319           | Airbus A319-100      |  6700
 SU9           | Sukhoi SuperJet-100 |  3300
 CN1           | Cessna 208 Caravan   |  2100
 CR2           | Bombardier CRJ-200   |  1900
(6 строк)
```

На втором терминале проведем ряд изменений:

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;

BEGIN
```

Добавим одну строку и одну строку обновим:

```
INSERT INTO aircrafts_tmp
  VALUES ( 'IL9', 'Ilyushin IL96', 9800 );

INSERT 0 1
```

```
UPDATE aircrafts_tmp
  SET range = range + 100
  WHERE aircraft_code = '320';

UPDATE 1
```

```
END;

COMMIT
```

Переходим на первый терминал.

```
SELECT *
  FROM aircrafts_tmp;
```

На первом терминале ничего не изменилось: фантомные строки не видны, и также не видны изменения в уже существующих строках. Это объясняется тем, что снимок данных выполняется на момент начала выполнения первого запроса транзакции.

```
 aircraft_code |        model         | range
---------------+----------------------+-------
 320           | Airbus A320-200      |  5700
 321           | Airbus A321-200      |  5600
 319           | Airbus A319-100      |  6700
 SU9           | Sukhoi SuperJet-100 |  3300
 CN1           | Cessna 208 Caravan   |  2100
 CR2           | Bombardier CRJ-200   |  1900
(6 строк)
```

Завершим первую транзакцию тоже:

```
END;

COMMIT
```

А теперь посмотрим, что изменилось в таблице:

```
SELECT *
  FROM aircrafts_tmp;
```

```
 aircraft_code |        model         | range
---------------+----------------------+-------
 321           | Airbus A321-200      |  5600
 319           | Airbus A319-100      |  6700
 SU9           | Sukhoi SuperJet-100 |  3300
 CN1           | Cessna 208 Caravan   |  2100
 CR2           | Bombardier CRJ-200   |  1900
 IL9           | Ilyushin IL96        |  9800
 320           | Airbus A320-200      |  5800
(7 строк)
```

Как видим, одна строка добавлена, а значение атрибута `range` у самолета Airbus A320-200 стало на 100 больше, чем было. Но до тех пор, пока мы на первом терминале находились в процессе выполнения первой транзакции, все эти изменения не были ей доступны, поскольку первая транзакция использовала снимок, сделанный до внесения изменений и их фиксации второй транзакцией.

Теперь покажем ошибки сериализации.

Начнем транзакцию на первом терминале:

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

```
BEGIN
```

```
UPDATE aircrafts_tmp
  SET range = range + 100
  WHERE aircraft_code = '320';
```

```
UPDATE 1
```

На втором терминале попытаемся обновить ту же строку:

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;

BEGIN
```

```
UPDATE aircrafts_tmp
  SET range = range + 200
  WHERE aircraft_code = '320';
```

Команда `UPDATE` на втором терминале ожидает завершения первой транзакции.

Перейдя на первый терминал, завершим первую транзакцию:

```
END;
```

```
COMMIT
```

Перейдя на второй терминал, увидим сообщение об ошибке:

```
ОШИБКА: не удалось сериализовать доступ из-за параллельного изменения
```

Поскольку обновление, произведенное в первой транзакции, не было зафиксировано на момент начала выполнения первого (и, в данном частном случае, единственного) запроса во второй транзакции, то возникает эта ошибка. Это объясняется вот чем. При выполнении обновления строки команда `UPDATE` во второй транзакции видит,

что строка уже изменена. На уровне изоляции Repeatable Read снимок данных создается на момент начала выполнения первого запроса транзакции и в течение транзакции уже не меняется, т. е. новая версия строки не считывается, как это делалось на уровне Read Committed. Но если выполнить обновление во второй транзакции без повторного считывания строки из таблицы, тогда будет иметь место потерянное обновление, что недопустимо. В результате генерируется ошибка, и вторая транзакция откатывается. Мы вводим команду END на втором терминале, но PostgreSQL выполняет не фиксацию (COMMIT), а откат:

```
END;

ROLLBACK
```

Если выполним запрос, то увидим, что было проведено только изменение в первой транзакции:

```
SELECT *
  FROM aircrafts_tmp
  WHERE aircraft_code = '320';

 aircraft_code |        model     | range
---------------+------------------+-------
 320           | Airbus A320-200 |  5900
(1 строка)
```

9.5. Уровень изоляции Serializable

Самый высший уровень изоляции транзакций — Serializable. Транзакции могут работать параллельно точно так же, как если бы они выполнялись последовательно одна за другой. Однако, как и при использовании уровня Repeatable Read, приложение должно быть готово к тому, что придется перезапускать транзакцию, которая была прервана системой из-за обнаружения зависимостей чтения/записи между транзакциями. Группа транзакций может быть параллельно выполнена и успешно зафиксирована в том случае, когда результат их параллельного выполнения был бы эквивалентен результату выполнения этих транзакций при выборе *одного из возможных вариантов* их упорядочения, если бы они выполнялись последовательно, одна за другой.

Для проведения эксперимента создадим специальную таблицу, в которой будет всего два столбца: один — числовой, а второй — текстовый. Назовем эту таблицу modes.

```
CREATE TABLE modes (
  num integer,
  mode text
);
```

```
CREATE TABLE
```

Добавим в таблицу две строки.

```
INSERT INTO modes VALUES ( 1, 'LOW' ), ( 2, 'HIGH' );
```

```
INSERT 0 2
```

Итак, содержимое таблицы имеет вид:

```
SELECT * FROM modes;
```

```
 num | mode
-----+------
   1 | LOW
   2 | HIGH
(2 строки)
```

На первом терминале начнем транзакцию и обновим одну строку из тех двух строк, которые были показаны в предыдущем запросе.

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

```
BEGIN
```

В команде обновления строки будем использовать предложение RETURNING. Поскольку значение поля num не изменяется, то будет видно, какая строка была обновлена. Это особенно пригодится во второй транзакции.

```
UPDATE modes
  SET mode = 'HIGH'
  WHERE mode = 'LOW'
  RETURNING *;
```

```
 num | mode
-----+------
   1 | HIGH
(1 строка)

UPDATE 1
```

На втором терминале тоже начнем транзакцию и обновим другую строку из тех двух строк, которые были показаны выше.

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN
```

```
UPDATE modes
  SET mode = 'LOW'
  WHERE mode = 'HIGH'
  RETURNING *;

 num | mode
-----+------
   2 | LOW
(1 строка)

UPDATE 1
```

Изменение, произведенное в первой транзакции, вторая транзакция не видит, поскольку на уровне изоляции Serializable каждая транзакция работает с тем снимком базы данных, который был сделан непосредственно перед выполнением ее первого оператора. Поэтому обновляется только одна строка, та, в которой значение поля mode было равно HIGH изначально.

Обратите внимание, что обе команды UPDATE были выполнены, ни одна из них не ожидает завершения другой транзакции.

Посмотрим, что получилось в первой транзакции:

```
SELECT * FROM modes;

 num | mode
-----+------
   2 | HIGH
   1 | HIGH
(2 строки)
```

А во второй транзакции:

```
SELECT * FROM modes;

 num | mode
-----+------
   1 | LOW
   2 | LOW
(2 строки)
```

Заканчиваем эксперимент. Сначала завершим транзакцию на первом терминале:

```
COMMIT;
```

```
COMMIT
```

А потом на втором терминале:

```
COMMIT;

ОШИБКА: не удалось сериализовать доступ из-за зависимостей чтения/записи
  между транзакциями
ПОДРОБНОСТИ: Reason code: Canceled on identification as a pivot, during
  commit attempt.
ПОДСКАЗКА: Транзакция может завершиться успешно при следующей попытке.
```

Какое же изменение будет зафиксировано? То, которое сделала транзакция, первой выполнившая фиксацию изменений.

```
SELECT * FROM modes;
```

```
 num | mode
-----+------
   2 | HIGH
   1 | HIGH
(2 строки)
```

Таким образом, параллельное выполнение двух транзакций сериализовать не удалось. Почему? Если обратиться к определению концепции сериализации, то нужно рассуждать так. Если бы была зафиксирована и вторая транзакция, тогда в таблице modes содержались бы такие строки:

```
 num | mode
-----+------
   1 | HIGH
   2 | LOW
```

Но этот результат не соответствует результату выполнения транзакций *ни при одном* из двух возможных вариантов их упорядочения, если бы они выполнялись последовательно. Следовательно, с точки зрения концепции сериализации эти транзакции невозможно сериализовать.

Покажем это, выполнив транзакции последовательно.

Предварительно необходимо пересоздать таблицу `modes` или с помощью команды `UPDATE` вернуть ее измененным строкам исходное состояние. Теперь обе транзакции можно выполнять на одном терминале. Первый вариант их упорядочения такой:

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN

UPDATE modes
  SET mode = 'HIGH'
  WHERE mode = 'LOW'
  RETURNING *;

 num | mode
-----+------
   1 | HIGH
(1 строка)

UPDATE 1

END;

COMMIT

BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN

UPDATE modes
  SET mode = 'LOW'
  WHERE mode = 'HIGH'
  RETURNING *;

 num | mode
-----+------
   2 | LOW
   1 | LOW
(2 строки)

UPDATE 2

END;

COMMIT
```

Проверим, что получилось:

```
SELECT * FROM modes;
```

```
 num | mode
-----+------
   2 | LOW
   1 | LOW
(2 строки)
```

Во втором варианте упорядочения поменяем транзакции местами. Конечно, предварительно нужно привести таблицу в исходное состояние.

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

```
BEGIN
```

```
UPDATE modes
  SET mode = 'LOW'
  WHERE mode = 'HIGH'
  RETURNING *;
```

```
 num | mode
-----+------
   2 | LOW
(1 строка)

UPDATE 1
```

```
END;
```

```
COMMIT
```

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

```
BEGIN
```

```
UPDATE modes
  SET mode = 'HIGH'
  WHERE mode = 'LOW'
  RETURNING *;
```

```
 num | mode
-----+------
   1 | HIGH
   2 | HIGH
(2 строки)

UPDATE 2

END;

COMMIT

SELECT * FROM modes;
```

Теперь результат отличается от того, который был получен при реализации первого варианта упорядочения транзакций.

```
 num | mode
-----+------
   1 | HIGH
   2 | HIGH
(2 строки)
```

Изменение порядка выполнения транзакций приводит к разным результатам. Однако если бы при параллельном выполнении транзакций была зафиксирована и вторая из них, то полученный результат не соответствовал бы *ни одному* из продемонстрированных возможных результатов последовательного выполнения транзакций. Таким образом, выполнить сериализацию этих транзакций невозможно. Обратите внимание, что вторая команда UPDATE в обоих случаях обновляет не одну строку, а две.

9.6. Пример использования транзакций

Продемонстрируем использование транзакций на примере базы данных «Авиаперевозки». Для этого создадим новое бронирование и оформим два билета с двумя перелетами в каждом. Выберем в качестве уровня изоляции Read Committed.

```
BEGIN;

BEGIN;
```

Сначала добавим запись в таблицу «Бронирования», причем назначим значение поля `total_amount` равным 0. После завершения ввода строк в таблицу «Перелеты» мы обновим это значение: оно станет равным сумме стоимостей всех забронированных перелетов. В качестве даты бронирования возьмем дату, которая была принята в качестве текущей в базе данных. Эту дату выдает функция `now`, созданная в схеме `bookings`.

```
INSERT INTO bookings ( book_ref, book_date, total_amount )
  VALUES ( 'ABC123', bookings.now(), 0 );
```

```
INSERT 0 1
```

Оформим два билета на двух разных пассажиров.

```
INSERT INTO tickets ( ticket_no, book_ref, passenger_id, passenger_name)
  VALUES ( '9991234567890', 'ABC123', '1234 123456', 'IVAN PETROV' );
```

```
INSERT 0 1
```

```
INSERT INTO tickets ( ticket_no, book_ref, passenger_id, passenger_name)
  VALUES ( '9991234567891', 'ABC123', '4321 654321', 'PETR IVANOV' );
```

```
INSERT 0 1
```

Отправим обоих пассажиров по маршруту Москва — Красноярск и обратно.

```
INSERT INTO ticket_flights
  ( ticket_no, flight_id, fare_conditions, amount )
  VALUES ( '9991234567890', 5572, 'Business', 12500 ),
         ( '9991234567890', 13881, 'Economy', 8500 );
```

```
INSERT 0 2
```

```
INSERT INTO ticket_flights
  ( ticket_no, flight_id, fare_conditions, amount )
  VALUES ( '9991234567891', 5572, 'Business', 12500 ),
         ( '9991234567891', 13881, 'Economy', 8500 );
```

```
INSERT 0 2
```

Подсчитаем общую стоимость забронированных билетов и запишем ее в строку таблицы «Бронирования». Конечно, если такая транзакция выполняется в рамках прикладной программы, то возможно, что подсчет общей суммы будет выполняться в этой программе. Тогда в команде `UPDATE` уже не потребуется выполнять подзапрос, а будет использоваться заранее вычисленное значение. Но более надежным решением было бы использование триггера для увеличения значения поля `total_amount`

при каждом добавлении строки в таблицу `ticket_flights`, но в этом учебном пособии они не рассматриваются.

```
UPDATE bookings
  SET total_amount =
    ( SELECT sum( amount )
        FROM ticket_flights
        WHERE ticket_no IN
        ( SELECT ticket_no
            FROM tickets
            WHERE book_ref = 'ABC123'
        )
    )
  WHERE book_ref = 'ABC123';
```

```
UPDATE 1
```

Проверим, что получилось.

```
SELECT *
  FROM bookings
  WHERE book_ref = 'ABC123';
```

```
 book_ref |        book_date       | total_amount
----------+------------------------+--------------
 ABC123   | 2016-10-13 22:00:00+08 |     42000.00
(1 строка)
```

```
COMMIT;
```

```
COMMIT;
```

В начале главы говорилось о свойствах транзакций. Их удобно прокомментировать на примере этой транзакции, в которой участвуют три таблицы. *Атомарность* говорит о том, что либо транзакция выполняется и фиксируется полностью, либо не фиксируется ни одна из ее операций. Поэтому в случае отказа сервера баз данных в процессе выполнения транзакции и последующего восстановления состояния базы данных те операции, которые уже были выполнены, будут отменены. Таким образом, база данных будет приведена к тому *согласованному* состоянию, в котором она находилась до начала транзакции. При выборе соответствующего уровня *изоляции* эта транзакция сможет выполняться, не подвергаясь помехам со стороны других параллельных транзакций. После успешной фиксации всех выполненных изменений в базе данных пользователь может быть уверен, что они станут *долговечными* и сохранятся даже в случае сбоя в работе сервера.

9.7. Блокировки

Кроме поддержки уровней изоляции транзакций, PostgreSQL позволяет также создавать явные блокировки данных как на уровне отдельных строк, так и на уровне целых таблиц. Блокировки могут быть востребованы при проектировании транзакций с уровнем изоляции, как правило, Read Committed, когда требуется более детальное управление параллельным выполнением транзакций. PostgreSQL предлагает много различных видов блокировок, но мы ограничимся рассмотрением только двух из них.

Команда SELECT имеет предложение FOR UPDATE, которое позволяет заблокировать отдельные строки таблицы с целью их последующего обновления. Если одна транзакция заблокировала строки с помощью этой команды, тогда параллельные транзакции не смогут заблокировать эти же строки до тех пор, пока первая транзакция не завершится, и тем самым блокировка не будет снята.

Проведем эксперимент, как и прежде, с использованием двух терминалов. Мы не будем приводить все вспомогательные команды создания и завершения транзакций, а ограничимся только командами, выполняющими полезную работу.

Итак, на первом терминале организуйте транзакцию с уровнем изоляции Read Committed и выполните следующую команду:

```
SELECT *
  FROM aircrafts_tmp
  WHERE model ~ '^Air'
  FOR UPDATE;
```

```
 aircraft_code |        model     | range
---------------+------------------+-------
 320           | Airbus A320-200 |  5700
 321           | Airbus A321-200 |  5600
 319           | Airbus A319-100 |  6700
(3 строки)
```

На втором терминале организуйте аналогичную транзакцию и выполните точно такую же команду. Вы увидите, что ее выполнение будет приостановлено.

```
SELECT *
  FROM aircrafts_tmp
  WHERE model ~ '^Air'
  FOR UPDATE;
```

На первом терминале обновите одну строку, а затем завершите транзакцию:

```
UPDATE aircrafts_tmp
  SET range = 5800
  WHERE aircraft_code = '320';
```

```
UPDATE 1
```

Перейдя на второй терминал, вы увидите, что там была, наконец, выполнена выборка, которая показала уже измененные данные:

```
 aircraft_code |        model    | range
---------------+-----------------+-------
 320           | Airbus A320-200 |  5800
 321           | Airbus A321-200 |  5600
 319           | Airbus A319-100 |  6700
(3 строки)
```

Завершите и вторую транзакцию.

Аналогичным образом можно организовать блокировки на уровне таблиц. Также на первом терминале организуйте транзакцию с уровнем изоляции Read Committed и выполните команду блокировки всей таблицы в самом строгом режиме, в котором другим транзакциям доступ к этой таблице запрещен полностью:

```
LOCK TABLE aircrafts_tmp
  IN ACCESS EXCLUSIVE MODE;
```

```
LOCK TABLE
```

На втором терминале выполните совершенно «безобидную» команду:

```
SELECT *
  FROM aircrafts_tmp
  WHERE model ~ '^Air';
```

Вы увидите, что выполнение команды SELECT на втором терминале будет задержано. Прервите транзакцию на первом терминале командой ROLLBACK. Вы увидите, что на втором терминале команда будет успешно выполнена.

Более подробно ознакомиться с различными видами блокировок уровня строки и уровня таблицы можно с помощью документации (раздел 13.3 «Явные блокировки»).

Контрольные вопросы и задания

1. По умолчанию каждая SQL-команда, выполняемая в среде psql, образует отдельную транзакцию с уровнем изоляции Read Committed. Поэтому в тех экспериментах, когда одна из транзакций состоит только из единственной SQL-команды, можно не выполнять команды `BEGIN` и `END`. Конечно, если каждая из параллельных транзакций состоит из единственной SQL-команды, то хотя бы для одной из транзакций придется все же выполнить и команду `BEGIN`, иначе эксперимент не получится.

   В тексте главы были приведены примеры транзакций, в которых рассматривались команды `SELECT ... FOR UPDATE` и `LOCK TABLE`. Попробуйте повторить эти эксперименты с учетом описанного поведения PostgreSQL.

2. Транзакции, работающие на уровне изоляции Read Committed, видят только свои собственные обновления и обновления, *зафиксированные* параллельными транзакциями. При этом нужно учитывать, что иногда могут возникать ситуации, которые на первый взгляд кажутся парадоксальными, но на самом деле все происходит в строгом соответствии с этим принципом.

   Воспользуемся таблицей «Самолеты» (`aircrafts`) или ее копией. Предположим, что мы решили удалить из таблицы те модели, дальность полета которых менее 2 000 км. В таблице представлена одна такая модель — Cessna 208 Caravan, имеющая дальность полета 1 200 км. Для выполнения удаления мы организовали транзакцию. Однако параллельная транзакция, которая, причем, началась раньше, успела обновить таблицу таким образом, что дальность полета самолета Cessna 208 Caravan стала составлять 2 100 км, а вот для самолета Bombardier CRJ-200 она, напротив, уменьшилась до 1 900 км. Таким образом, в результате выполнения операций обновления в таблице по-прежнему присутствует строка, удовлетворяющая первоначальному условию, т. е. значение атрибута `range` у которой меньше 2000.

   Наша задача: проверить, будет ли в результате выполнения двух транзакций удалена какая-либо строка из таблицы.

   На первом терминале начнем транзакцию, при этом уровень изоляции Read Committed в команде указывать не будем, т. к. он принят по умолчанию:

   ```
   BEGIN;

   BEGIN
   ```

```
SELECT *
  FROM aircrafts_tmp
  WHERE range < 2000;
```

```
 aircraft_code |        model       | range
---------------+--------------------+-------
 CN1           | Cessna 208 Caravan |  1200
(1 строка)
```

```
UPDATE aircrafts_tmp
  SET range = 2100
  WHERE aircraft_code = 'CN1';
```

```
UPDATE 1
```

```
UPDATE aircrafts_tmp
  SET range = 1900
  WHERE aircraft_code = 'CR2';
```

```
UPDATE 1
```

На втором терминале начнем вторую транзакцию, которая и будет пытаться удалить строки, у которых значение атрибута `range` меньше 2000.

```
BEGIN;

BEGIN
```

```
SELECT *
  FROM aircrafts_tmp
  WHERE range < 2000;

 aircraft_code |        model       | range
---------------+--------------------+-------
 CN1           | Cessna 208 Caravan |  1200
(1 строка)
```

```
DELETE FROM aircrafts_tmp WHERE range < 2000;
```

Введя команду `DELETE`, мы видим, что она не завершается, а ожидает, когда со строки, подлежащей удалению, будет снята блокировка. Блокировка, установленная командой `UPDATE` в первой транзакции, снимается только при завершении транзакции, а завершение может иметь два исхода: фиксацию изменений с помощью команды `COMMIT` (или `END`) или отмену изменений с помощью команды `ROLLBACK`.

Давайте зафиксируем изменения, выполненные первой транзакцией. На первом терминале сделаем так:

```
COMMIT;

COMMIT
```

Тогда на втором терминале мы получим такой результат от команды DELETE:

```
DELETE 0
```

Чем объясняется такой результат? Он кажется нелогичным: ведь команда SELECT, выполненная в этой же второй транзакции, показывала наличие строки, удовлетворяющей условию удаления.

Объяснение таково: поскольку вторая транзакция пока еще не видит изменений, произведенных в первой транзакции, то команда DELETE выбирает для удаления строку, описывающую модель Cessna 208 Caravan, однако эта строка была заблокирована в первой транзакции командой UPDATE. Эта команда изменила значение атрибута range в этой строке.

При завершении первой транзакции блокировка с этой строки снимается (со второй строки — тоже), и команда DELETE во второй транзакции получает возможность заблокировать эту строку. При этом команда DELETE данную строку *перечитывает* и вновь вычисляет условие WHERE применительно к ней. Однако теперь условие WHERE для данной строки уже не выполняется, следовательно, эту строку удалять нельзя. Конечно, в таблице есть теперь другая строка, для самолета Bombardier CRJ-200, удовлетворяющая условию удаления, однако повторный поиск строк, удовлетворяющих условию WHERE в команде DELETE, не производится.

В результате не удаляется ни одна строка. Таким образом, к сожалению, имеет место нарушение согласованности, которое можно объяснить деталями реализации СУБД.

Завершим вторую транзакцию:

```
END;

COMMIT
```

Вот что получилось в результате:

```
SELECT * FROM aircrafts_tmp;
```

```
 aircraft_code |        model         | range
---------------+----------------------+-------
 773           | Boeing 777-300       | 11100
 763           | Boeing 767-300       |  7900
 SU9           | Sukhoi SuperJet-100  |  3000
 320           | Airbus A320-200      |  5700
 321           | Airbus A321-200      |  5600
 319           | Airbus A319-100      |  6700
 733           | Boeing 737-300       |  4200
 CN1           | Cessna 208 Caravan   |  2100
 CR2           | Bombardier CRJ-200   |  1900
(9 строк)
```

Задание. Модифицируйте сценарий выполнения транзакций: в первой транзакции вместо фиксации изменений выполните их отмену с помощью команды ROLLBACK и посмотрите, будет ли удалена строка и какая конкретно.

3.* Когда говорят о таком феномене, как потерянное обновление, то зачастую в качестве примера приводится операция UPDATE, в которой значение какого-то атрибута изменяется с применением одного из действий арифметики. Например:

```
UPDATE aircrafts_tmp
  SET range = range + 200
  WHERE aircraft_code = 'CR2';
```

При выполнении двух и более подобных обновлений в рамках параллельных транзакций, использующих, например, уровень изоляции Read Committed, будут учтены все такие изменения (что и было показано в тексте главы). Очевидно, что потерянного обновления не происходит.

Предположим, что в одной транзакции будет просто присваиваться новое значение, например, так:

```
UPDATE aircrafts_tmp
  SET range = 2100
  WHERE aircraft_code = 'CR2';
```

А в параллельной транзакции будет выполняться аналогичная команда:

```
UPDATE aircrafts_tmp
  SET range = 2500
  WHERE aircraft_code = 'CR2';
```

Очевидно, что сохранится только одно из значений атрибута `range`. Можно ли говорить, что в такой ситуации имеет место потерянное обновление? Если оно имеет место, то что можно предпринять для его недопущения? Обоснуйте ваш ответ.

Для получения дополнительной информации можно обратиться к фундаментальному труду К. Дж. Дейта, а также к полному руководству по SQL Дж. Гроффа, П. Вайнберга и Э. Оппеля. Библиографические описания этих книг приведены в списке рекомендуемой литературы.

4. На уровне изоляции транзакций Read Committed имеет место такой феномен, как чтение фантомных строк. Такие строки могут появляться в выборке как в результате добавления новых строк параллельной транзакцией, так и вследствие изменения ею значений атрибутов, участвующих в формировании условия выборки. Рассмотрим пример, иллюстрирующий вторую из указанных причин.

   На первом терминале организуем транзакцию. Она будет иметь уровень изоляции Read Committed:

```
BEGIN;

BEGIN

SELECT *
  FROM aircrafts_tmp
  WHERE range > 6000;

 aircraft_code |       model     | range
---------------+-----------------+-------
 773           | Boeing 777-300  | 11100
 763           | Boeing 767-300  |  7900
 319           | Airbus A319-100 |  6700
(3 строки)
```

   На втором терминале организуем транзакцию и обновим одну из строк таблицы таким образом, чтобы эта строка стала удовлетворять условию отбора строк, заданному в первой транзакции.

```
BEGIN;

BEGIN
```

```
UPDATE aircrafts_tmp
  SET range = 6100
  WHERE aircraft_code = '320';

UPDATE 1
```

Сразу завершим вторую транзакцию, чтобы первая транзакция увидела эти изменения.

```
END;

COMMIT
```

На первом терминале повторим ту же самую выборку:

```
SELECT *
  FROM aircrafts_tmp
  WHERE range > 6000;

 aircraft_code |        model     | range
---------------+------------------+-------
 773           | Boeing 777-300   | 11100
 763           | Boeing 767-300   |  7900
 319           | Airbus A319-100  |  6700
 320           | Airbus A320-200  |  6100
(4 строки)
```

Транзакция еще не завершилась, но она уже увидела новую строку, обновленную зафиксированной параллельной транзакцией. Теперь эта строка стала соответствовать условию выборки. Таким образом, не изменяя критерий выборки, мы получили другое множество строк.

Завершим теперь и первую транзакцию:

```
END;

COMMIT
```

Задание. Модифицируйте этот эксперимент: вместо операции `UPDATE` используйте операцию `INSERT`.

5. В тексте главы была рассмотрена команда `SELECT ... FOR UPDATE`, выполняющая блокировку на уровне отдельных строк. Организуйте две параллельные

транзакции с уровнем изоляции Read Committed и выполните с ними ряд экспериментов. В первой транзакции заблокируйте некоторое множество строк, отбираемых с помощью условия `WHERE`. А во второй транзакции изменяйте условие выборки таким образом, чтобы выбираемое множество строк:

- являлось подмножеством множества строк, выбираемых в первой транзакции;
- являлось надмножеством множества строк, выбираемых в первой транзакции;
- пересекалось с множеством строк, выбираемых в первой транзакции;
- не пересекалось с множеством строк, выбираемых в первой транзакции.

Наблюдайте за поведением команд выборки в каждой транзакции. Попробуйте обобщить ваши наблюдения.

6. Самостоятельно ознакомьтесь с предложением `FOR SHARE` команды `SELECT` и выполните необходимые эксперименты. Используйте документацию: раздел 13.3.2 «Блокировки на уровне строк» и описание команды `SELECT`.

7. В тексте главы для иллюстрации изучаемых концепций мы создавали только две параллельные транзакции. Попробуйте воспроизвести представленные эксперименты, создав три или даже четыре параллельные транзакции.

8.* В тексте главы была рассмотрена транзакция для выполнения бронирования билетов. Для нее был выбран уровень изоляции Read Committed.

   Как вы думаете, если одновременно будут производиться несколько операций бронирования, то, может быть, имеет смысл «ужесточить» уровень изоляции до Serializable? Или нет необходимости это делать? Обдумайте и вариант с использованием явных блокировок. Обоснуйте ваш ответ.

9.* В разделе документации 13.2.3 «Уровень изоляции Serializable» сказано, что если поиск в таблице осуществляется последовательно, без использования индекса, тогда на всю таблицу накладывается так называемая предикатная блокировка. Такой подход приводит к увеличению числа сбоев сериализации. В качестве контрмеры можно попытаться использовать индексы. Конечно, если таблица совсем небольшая, то может и не получиться заставить PostgreSQL использовать поиск по индексу. Тем не менее давайте выполним следующий эксперимент.

   Для его проведения создадим специальную таблицу, в которой будет всего два столбца: один — числовой, а второй — текстовый. Значения во втором столбце будут иметь вид: `LOW1`, `LOW2`, ..., `HIGH1`, `HIGH2`, ... Назовем эту таблицу `modes`.

Добавим в нее такое число строк, которое сделает очень вероятным использование индекса при выполнении операций обновления строк и, соответственно, отсутствие предикатной блокировки всей таблицы. О том, как узнать, используется ли индекс при выполнении тех или иных операций, написано в главе 10.

```
CREATE TABLE modes AS
  SELECT num::integer, 'LOW' || num::text AS mode
    FROM generate_series( 1, 100000 ) AS gen_ser( num )
  UNION ALL
  SELECT num::integer, 'HIGH' || ( num - 100000 )::text AS mode
    FROM generate_series( 100001, 200000 ) AS gen_ser( num );
```

```
SELECT 200000
```

Проиндексируем таблицу по числовому столбцу.

```
CREATE INDEX modes_ind
  ON modes ( num );
```

```
CREATE INDEX
```

Из всего множества строк нас будут интересовать только две:

```
SELECT *
  FROM modes
  WHERE mode IN ( 'LOW1', 'HIGH1' );
```

```
  num   | mode
--------+-------
      1 | LOW1
 100001 | HIGH1
(2 строки)
```

На первом терминале начнем транзакцию и обновим одну строку из тех двух строк, которые были показаны в предыдущем запросе.

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

```
BEGIN
```

```
UPDATE modes
  SET mode = 'HIGH1'
  WHERE num = 1;
```

```
UPDATE 1
```

На втором терминале тоже начнем транзакцию и обновим другую строку из тех двух строк, которые были показаны выше.

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN
```

```
UPDATE modes
  SET mode = 'LOW1'
  WHERE num = 100001;

UPDATE 1
```

Обратите внимание, что обе команды UPDATE были выполнены, ни одна из них не ожидает завершения другой транзакции.

Попробуем завершить транзакции. Сначала — на первом терминале:

```
COMMIT;

COMMIT
```

А потом на втором терминале:

```
COMMIT;

COMMIT
```

Посмотрим, что получилось:

```
SELECT *
  FROM modes
  WHERE mode IN ( 'LOW1', 'HIGH1' );

  num   | mode
--------+-------
      1 | HIGH1
 100001 | LOW1
(2 строки)
```

Теперь система смогла сериализовать параллельные транзакции и зафиксировать их обе. Как вы думаете, почему это удалось? Обосновывая ваш ответ, примите во внимание тот результат, который был бы получен при последовательном выполнении транзакций.

10.* В тексте главы был рассмотрен пример транзакции над таблицами базы данных «Авиаперевозки». Давайте теперь создадим две параллельные транзакции и выполним их с уровнем изоляции Serializable. Отправим также двоих пассажиров теми же самыми рейсами, что и ранее, но операции распределим между двумя транзакциями. Отличие заключается в том, что в начале транзакции будут выполняться выборки из таблицы `ticket_flights`. Для упрощения ситуации не будем предварительно проверять наличие свободных мест, т. к. сейчас для нас важно не это. Итак, первая транзакция:

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN

SELECT *
  FROM ticket_flights
  WHERE flight_id = 13881;

   ticket_no   | flight_id | fare_conditions |  amount
---------------+-----------+-----------------+----------
 0005433848165 |     13881 | Business        | 99800.00
 ...
 0005433848007 |     13881 | Economy         | 33300.00
(82 строки)

INSERT INTO bookings ( book_ref, book_date, total_amount )
  VALUES ( 'ABC123', bookings.now(), 0 );

INSERT 0 1

INSERT INTO tickets
  ( ticket_no, book_ref, passenger_id, passenger_name )
  VALUES ( '9991234567890', 'ABC123', '1234 123456', 'IVAN PETROV' );

INSERT 0 1

INSERT INTO ticket_flights
  ( ticket_no, flight_id, fare_conditions, amount )
  VALUES ( '9991234567890', 13881, 'Business', 12500 );

INSERT 0 1

UPDATE bookings
  SET total_amount = 12500
  WHERE book_ref = 'ABC123';

UPDATE 1
```

```
COMMIT;
```

```
COMMIT
```

Вторая транзакция:

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

```
BEGIN
```

```
SELECT *
  FROM ticket_flights
  WHERE flight_id = 5572;
```

```
  ticket_no    | flight_id | fare_conditions |  amount
---------------+-----------+-----------------+----------
 0005433847924 |      5572 | Business        | 99800.00
 ...
 0005433847890 |      5572 | Economy         | 33300.00
(100 строк)
```

```
INSERT INTO bookings ( book_ref, book_date, total_amount )
  VALUES ( 'ABC456', bookings.now(), 0 );
```

```
INSERT 0 1
```

```
INSERT INTO tickets
  ( ticket_no, book_ref, passenger_id, passenger_name )
  VALUES ( '9991234567891', 'ABC456', '4321 654321', 'PETR IVANOV' );
```

```
INSERT 0 1
```

```
INSERT INTO ticket_flights
  ( ticket_no, flight_id, fare_conditions, amount )
  VALUES ( '9991234567891', 5572, 'Business', 12500 );
```

```
INSERT 0 1
```

```
UPDATE bookings
  SET total_amount = 12500
  WHERE book_ref = 'ABC456';
```

```
UPDATE 1
```

```
COMMIT;

ОШИБКА: не удалось сериализовать доступ из-за зависимостей
  чтения/записи между транзакциями
ПОДРОБНОСТИ: Reason code: Canceled on identification as a pivot,
  during commit attempt.
ПОДСКАЗКА: Транзакция может завершиться успешно при следующей
  попытке.
```

Задание 1. Попытайтесь объяснить, почему транзакции не удалось сериализовать. Что можно сделать, чтобы удалось зафиксировать обе транзакции? Одно из возможных решений — понизить уровень изоляции. Другим решением может быть создание индекса по столбцу `flight_id` для таблицы `ticket_flights`. Почему создание индекса может помочь? Обратитесь за разъяснениями к разделу документации 13.2.3 «Уровень изоляции Serializable».

Задание 2. В первой транзакции условие в команде `SELECT` такое: `... WHERE flight_id = 13881`. В команде вставки в таблицу `ticket_flights` значение поля `flight_id` также равно `13881`. Во второй транзакции в этих же командах используется значение `5572`. Поменяйте местами значения в командах `SELECT` и повторите эксперименты, выполнив транзакции параллельно с уровнем изоляции Serializable. Почему сейчас наличие индекса не помогает зафиксировать обе транзакции? Вспомните, что аномалия сериализации — это ситуация, когда параллельное выполнение транзакций приводит к результату, невозможному ни при каком из вариантов упорядочения этих же транзакций при их последовательном выполнении.