PostgresPro

Язык SQL

Лекция 7
Транзакции

Е. П. Моргунов
Сибирский государственный университет науки и технологий
имени академика М. Ф. Решетнева
г. Красноярск
Институт информатики и телекоммуникаций
emorgunov@mail.ru

Компания Postgres Professional
г. Москва

Версия от 28.08.2019

Подготовка к работе (1)

На вашем компьютере уже должна быть развернута база данных demo.

- Войдите в систему как пользователь postgres:

```
su - postgres
```

- Должен быть запущен сервер баз данных PostgreSQL.

```
pg_ctl  start  -D /usr/local/pgsql/data  -l postgres.log
```

- Для проверки запуска сервера выполните команду

```
pg_ctl  status  -D /usr/local/pgsql/data
```

или

```
ps -ax | grep postgres | grep -v grep
```

- Запустите утилиту psql и подключитесь к базе данных demo

`psql -d demo -U postgres` (можно просто `psql -d demo`)

- Назначьте схему bookings в качестве текущей

```
demo=# set search_path = bookings;
```

Подготовка к работе (2)

- Для останова сервера баз данных PostgreSQL служит команда

```
pg_ctl  stop  -D /usr/local/pgsql/data  -l postgres.log
```

- Если у вас база данных demo была модифицирована, то для ее восстановления выполните команду

```
psql -f demo_small.sql -U postgres
```

7.1. Общая информация

Транзакция — это ...

PostgresPro

- 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.
- 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)**.
- Современные СУБД предлагают специальные механизмы для организации параллельного, т. е. одновременного, выполнения транзакций.

Транзакция — пример

PostgresPro

Процедура 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.

Реализация транзакций в СУБД PostgreSQL (1)

PostgresPro

- 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.
- Снимок – это не физическая копия всей базы данных, это несколько чисел, которые идентифицируют текущую транзакцию и те транзакции, которые уже выполнялись в момент начала текущей.
- При этом параллельно исполняемые транзакции, даже вносящие изменения в базу данных, не нарушают согласованности данных этого снимка.

Реализация транзакций в СУБД PostgreSQL (2)

PostgresPro

- 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.

Важное следствие применения MVCC — операции чтения никогда не блокируются операциями записи, а операции записи никогда не блокируются операциями чтения.

Подробно о многоверсионной модели см. учебный курс DBA1 компании Postgres Professional https://postgrespro.ru/education/courses/DBA1

Свойства транзакций согласно теории баз данных

PostgresPro

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.

ACID

Феномены при параллельном выполнении транзакций (1)

PostgresPro

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.

Феномены при параллельном выполнении транзакций (2)

4. **Фантомное чтение (phantom read).** Транзакция выполняет повторную выборку множества 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.

Смысл концепции сериализации транзакций (1)

- 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.

Смысл концепции сериализации транзакций (2)

PostgresPro

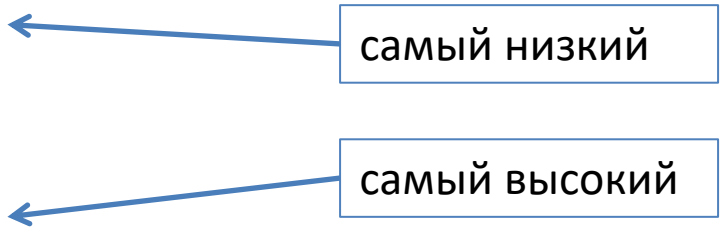
- Сериализация двух транзакций при их параллельном выполнении означает, что полученный результат будет соответствовать *одному из двух возможных* вариантов упорядочения транзакций при их последовательном выполнении.
- При этом нельзя сказать точно, какой из вариантов будет реализован.
- Если параллельно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

Уровни изоляции транзакций (1)

PostgresPro

- 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:
  - READ UNCOMMITTED
  - READ COMMITTED
  - REPEATABLE READ
  - SERIALIZABLE.
- Каждый более высокий уровень включает в себя все возможности предыдущего.



Уровни изоляции транзакций (2)

PostgresPro

- **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.
- **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.

Уровни изоляции транзакций (3)

PostgresPro

- **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.
- **SERIALIZABLE**. Не допускается ни один из феноменов, перечисленных выше, в том числе и аномалии сериализации.

Кто обеспечивает изоляцию транзакций?

PostgresPro

- 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)
```

7.2. Уровень изоляции READ UNCOMMITTED

Видит ли транзакция «грязные» данные ?

PostgresPro

- 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 со строками из таблицы aircrafts по внешнему ключу.

```
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.

Эксперимент (1)

PostgresPro

- На первом терминале выполним следующие команды:

1

```
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)
```

назначим уровень изоляции

изменение есть

Эксперимент (2)

PostgresPro

- Начнем транзакцию на втором терминале:

2

```
BEGIN;
BEGIN
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT * FROM aircrafts_tmp WHERE aircraft_code = 'SU9';
 aircraft_code |        model         | range
---------------+----------------------+-------
 SU9           | Sukhoi SuperJet-100  |  3000
(1 строка)
```

изменений нет

- 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.

Эксперимент (3)

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

1

```
ROLLBACK;
ROLLBACK
```

- На втором терминале сделаем так же:

2

```
ROLLBACK;
ROLLBACK
```

7.3. Уровень изоляции READ COMMITTED

Возможны ли потерянные изменения ?

PostgresPro

- 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.

Эксперимент (1)

- На первом терминале выполним следующие команды:

1

```
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 включили указание уровня изоляции непосредственно в команду BEGIN. Можно вообще было ограничиться только командой BEGIN.

= 3000 + 100

Транзакция видит незафиксированные изменения, выполненные в ней самой

Эксперимент (2)

- Во второй транзакции (на втором терминале) 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.

2

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

Команда перешла в состояние ожидания, поскольку команда UPDATE в первой транзакции заблокировала строку, а блокировка снимается только при завершении транзакции.

- Завершим первую транзакцию с фиксацией изменений:

1

```
COMMIT;
COMMIT
```

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

2

```
UPDATE 1
```

Эксперимент (3)

- 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)
```

= 3000 + 100 + 200

2

- 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:

```
END;
COMMIT
```

= COMMIT (расширение PostgreSQL)

На уровне изоляции READ UNCOMMITTED эффекта потерянных обновлений также не возникает

Эффект неповторяющегося чтения (1)

PostgresPro

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

1

```
BEGIN;
BEGIN
SELECT * FROM aircrafts_tmp;
 aircraft_code |         model          | range
---------------+------------------------+-------
 773           | Boeing 777-300         | 11100
 763           | Boeing 767-300         |  7900
 SU9           | Sukhoi SuperJet-100    |  3300
 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
(9 строк)
```

Эффект неповторяющегося чтения (2)

PostgresPro

- На втором терминале:

2

```
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 с фиксацией изменений:

```
END;
COMMIT
```

Эффект неповторяющегося чтения (3)

PostgresPro

- Повторим выборку в первой транзакции:

1

```
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
```

7.4. Уровень изоляции REPEATABLE READ

Общее описание этого уровня изоляции

PostgresPro

- Само его название говорит о том, что он не допускает наличия феномена 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.

Эксперимент (1)

PostgresPro

- На первом терминале:

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

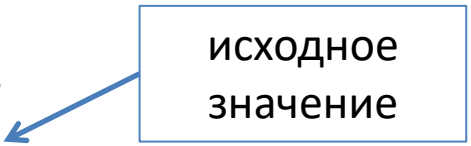
- Сначала посмотрим содержимое таблицы:

```
SELECT * FROM aircrafts_tmp;
```

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

1

```
 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)
```



Эксперимент (2)

- На втором терминале проведем ряд изменений и сразу завершим транзакцию:

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

- Добавим одну строку:

```
INSERT INTO aircrafts_tmp
VALUES ( 'IL9', 'Ilyushin IL96', 9800 );
INSERT 0 1
```

2

- А одну строку обновим:

```
UPDATE aircrafts_tmp SET range = range + 100
WHERE aircraft_code = '320';
UPDATE 1
END;
COMMIT
```

транзакция зафиксирована

Эксперимент (3)

PostgresPro

- 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.

1

```
 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)
```

неизмененное значение

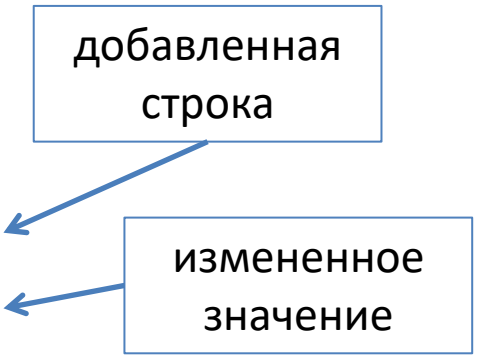
- Завершим первую транзакцию тоже:

```
END;
COMMIT
```

Эксперимент (4)

- 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)
```



- 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.

Эксперимент: ошибки сериализации (1)

PostgresPro

- Начнем транзакцию на первом терминале:

1
```
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:

2
```
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:

1
```
END;
COMMIT
```

Эксперимент : ошибки сериализации (2)

PostgresPro

2

- 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.

Эксперимент: ошибки сериализации (3)

PostgresPro

- Мы вводим команду END на втором терминале, но PostgreSQL выполняет не фиксацию (COMMIT), а откат:

2

```
END;
ROLLBACK
```

отмена изменений

- 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)
```

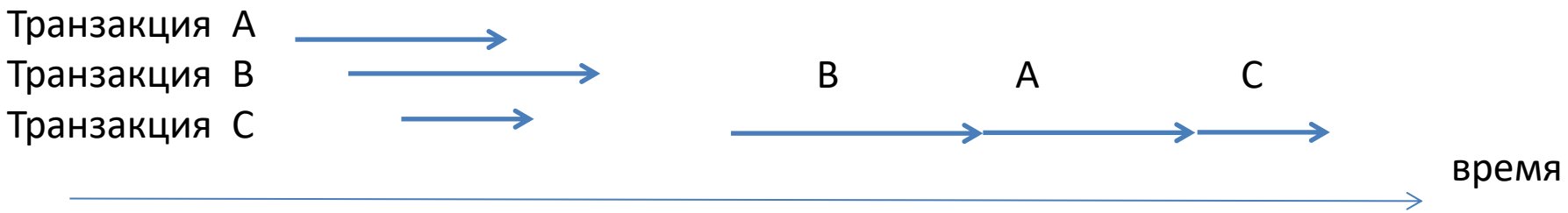
= 5800 + 100

7.5. Уровень изоляции SERIALIZABLE

Общее описание уровня изоляции

PostgresPro

- 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.



Подготовка к эксперименту

PostgresPro

- 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)
```

Эксперимент (1)

PostgresPro

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

1

```
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
```

Эксперимент (2)

PostgresPro

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

2

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN
UPDATE modes SET mode = 'LOW' WHERE mode = 'HIGH'
RETURNING *;
```

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

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

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

Эксперимент (3)

PostgresPro

- Посмотрим, что получилось в первой транзакции:

1

```
SELECT * FROM modes;
 num | mode
-----+------
   2 | HIGH
   1 | HIGH
(2 строки)
```

- А во второй транзакции:

2

```
SELECT * FROM modes;
 num | mode
-----+------
   1 | LOW
   2 | LOW
(2 строки)
```

Эксперимент (4)

- Заканчиваем эксперимент. Сначала завершим транзакцию на первом терминале:

1
```
COMMIT;
COMMIT
```

- А потом на втором терминале:

2
```
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.
```

Эксперимент (5)

- 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

Эксперимент: продолжение

PostgresPro

- 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.

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

Эксперимент: вариант 1

PostgresPro

```
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
```

две строки

```
SELECT * FROM modes;
 num | mode
-----+------
   2 | LOW
   1 | LOW
(2 строки)
```

результат

Эксперимент: вариант 2

```
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
```

две строки

```
SELECT * FROM modes;
 num | mode
-----+------
   1 | HIGH
   2 | HIGH
(2 строки)
```

результат

Вывод после завершения эксперимента

PostgresPro

- 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.

7.6. Пример использования транзакций

- Создадим новое бронирование и оформим два билета с двумя перелетами в каждом.

```
BEGIN;
BEGIN;
```

уровень изоляции READ COMMITTED

- Сначала добавим запись в таблицу «Бронирования», причем, значение поля total_amount назначим равным 0.

```
INSERT INTO bookings ( book_ref, book_date, total_amount )
VALUES ( 'ABC123', bookings.now(), 0 );
INSERT 0 1
```

Текущая дата в базе данных. Эту дату выдает функция now(), созданная в схеме bookings.

После завершения ввода строк в таблицу «Перелеты» мы обновим это значение: оно станет равным сумме стоимостей всех забронированных перелетов.

Оформим два билета

PostgresPro

- 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
```

Оформим конкретные перелеты

PostgresPro

- 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 билетов

PostgresPro

- Подсчитаем общую стоимость забронированных билетов и запишем ее в строку таблицы «Бронирования».
- Конечно более надежным решением было бы использование **триггера** для увеличения значения поля total_amount при каждом добавлении строки в таблицу ticket_flights.

```
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
```

Этот подзапрос коррелированный или нет?

Что получилось ?

- 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;
```

7.7. Блокировки

Общие сведения

- 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.

Эксперимент: блокировка строк (1)

- 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:

1

```
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)
```

Эксперимент: блокировка строк (2)

- На втором терминале организуйте аналогичную транзакцию и выполните точно такую же команду.

2
```
SELECT * FROM aircrafts_tmp
WHERE model ~ '^Air' FOR UPDATE;
```

Выполнение команды будет приостановлено.

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

1
```
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:

2
```
 aircraft_code |       model       | range
---------------+-------------------+-------
 320           | Airbus A320-200   |  5800
 321           | Airbus A321-200   |  5600
 319           | Airbus A319-100   |  6700
(3 строки)
```

измененное значение

- Завершите и вторую транзакцию.

Блокировки на уровне таблиц

PostgresPro

- 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:

1
```
LOCK TABLE aircrafts_tmp IN ACCESS EXCLUSIVE MODE;
LOCK TABLE
```

- На втором терминале выполните совершенно «безобидную» команду:

2
```
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 см. раздел документации 13.3 «Явные блокировки».

Литература

1. Грофф, Дж. SQL. Полное руководство : пер. с англ. / Джеймс Р. Грофф, Пол Н. Вайнберг, Эндрю Дж. Оппель. – 3-е изд. – М. : Вильямс, 2015. – 960 с.
2. Лузанов, П. В. Postgres. Первое знакомство [Текст] / П. В. Лузанов, Е. В. Рогов, И. В. Лёвшин. – 5-е изд., перераб. и доп. – М. : Постгрес Профессиональный, 2019. – 156 с. https://edu.postgrespro.ru/introbook_v5.pdf
3. Моргунов, Е. П. PostgreSQL. Основы языка SQL [Текст] : учеб. пособие / Е. П. Моргунов ; под ред. Е. В. Рогова, П. В. Лузанова. – СПб. : БХВ-Петербург, 2018. – 336 с. https://edu.postgrespro.ru/sql_primer.pdf
4. Новиков, Б. А. Основы технологий баз данных [Текст] : учеб. пособие / Б. А. Новиков, Е. А. Горшкова ; под ред. Е. В. Рогова. – М. : ДМК Пресс, 2019. – 240 с. https://edu.postgrespro.ru/dbtech_part1.pdf
5. Учебные курсы по администрированию PostgreSQL / Е. В. Рогов, П. В. Лузанов ; Postgres Professional. – https://postgrespro.ru/education/courses.
6. PostgreSQL [Электронный ресурс] : официальный сайт / The PostgreSQL Global Development Group. – https://www.postgresql.org.
7. Postgres Professional [Электронный ресурс] : российский производитель СУБД Postgres Pro : официальный сайт / Postgres Professional. – https://postgrespro.ru.

Задание

Для выполнения практических заданий необходимо использовать книгу:

Моргунов, Е. П. PostgreSQL. Основы языка SQL [Текст] : учеб. пособие / Е. П. Моргунов ; под ред. Е. В. Рогова, П. В. Лузанова. – СПб. : БХВ-Петербург, 2018. – 336 с.

https://postgrespro.ru/education/books/sqlprimer

1. Изучить материал главы 9. Запросы к базе данных выполнять с помощью утилиты psql, описанной в главе 2, параграф 2.2.