Лекция 12. Транзакции. Конкурентный доступ к данным

Азаревич Артём

30 ноября 2023 г.

Транзакция

Транзакция - атомарная единица работы с базой данный.

Когда транзакция изменяет БД, либо применяются все изменения, либо ни одного.

Зачем нужны транзакции?

Классический пример - двойной учёт.

Перевод $1000 от account id=1 к account id=2.

1. `UPDATE accounts SET amount = amount - 1000 WHERE id = 1`
2. `UPDATE accounts SET amount = amount + 1000 WHERE id = 2`

Что может пойти не так?

Зачем нужны транзакции? (2)

1. `UPDATE accounts SET amount = amount - 1000 WHERE id = 1`
   Ошибка
2. `UPDATE accounts SET amount = amount + 1000 WHERE id = 2`

У id=1 деньги отняли, у id=2 деньги не пришли. Потеряли $1000.

Решение - обернуть обе операции в транзакцию.

ACID. Atomicity

Определение тразакции в реляционных СУБД имеет 4 требования (ACID):

- Atomicity (Атомарность)
- Consistency (Согласованность)
- Isolation (Изоляция)
- Durability (Устойчивость)

Atomicity (атомарность) - любая транзакция будет зафиксирована (COMMIT) в системе полностью. Если одна из операций в последовательности не выполнена, производится откат (ROLLBACK), и БД остается во внешне исходном состоянии (т.е. с точки зрения пользователя БД никакой транзакции не было).

ACID. Consistency

Consistency (согласованность) - в каждый момент времени БД остается в согласованном состоянии.

Транзакция может перевести базу из только из одного согласованного состояния в другой - с соблюдением всех ограничений (CONSTRAINT), каскадных обновлений, триггеров, проверок и т.п. Нельзя ”сломать” БД неверной транзакцией.

Запросы к базе данных могут вернуть или старые, или новые данные, но не смесь.

ACID. Isolation

Isolation (изоляция) - транзакции изолированы друг от друга. Параллельные тразакции не должны оказывать влияние на результат данной транзакции.

Например, транзакциям недоступны незакомиченные данные друг друга.

На практике полная изоляция труднодостижима, поэтому обычно СУБД разрешают выбирать уровень изоляции (isolation level). Каждый уровень - компромисс в пользу большей защиты или большей производительности.

ACID. Durability

Durability (устойчивость) - если транзакция применена (и мы получили подтверждение транзакции), то ”назад пути нет”.

Если произойдет перебой питания, ошибка в приложении и т.п., изменения, примененные в транзакции, не могут потеряться.

Использование транзакций (1)

Начало явной транзакции - `START TRANSACTION`

Применение изменений - `COMMIT`

Откат изменений - `ROLLBACK`

Настройки транзакций - `SET TRANSACTION`

```
1  START TRANSACTION;
2  UPDATE accounts SET amount = amount - 1000 WHERE id = 1;
3  UPDATE accounts SET amount = amount + 1000 WHERE id = 2;
4  COMMIT;
```

https://www.postgresql.org/docs/current/tutorial-transactions.html

Использование транзакций (2)

В PostgreSQL, вся активность пользователя происходит внутри транзакции.

Т.е. по умолчанию каждая команда в сессии образует транзакцию из одного элемента.

Это можно отключить: `SET autocommit = 0`

```
1  SET autocommit = off;
2  UPDATE accounts SET amount = amount - 1000 WHERE id = 1;
3  UPDATE accounts SET amount = amount + 1000 WHERE id = 2;
4  COMMIT;
```

https://www.postgresql.org/docs/current/ecpg-sql-set-autocommit.html

Использование транзакций (3)

В других СУБД некоторые команды могут вызывать неявный коммит. Например, MySQL:

- все команды DDL (`CREATE TABLE`, `ALTER TABLE` и т.п.)
- `ALTER USER`, `CREATE USER`, `GRANT`...
- `SET autocommit = 1`
- `LOCK TABLES`, `UNLOCK TABLES`
- ...

В PostgreSQL возможно исполнять DDL в рамках транзакции (напр. добавить столбец и заполнить его данными). Но в PostgreSQL нельзя:

- `CREATE DATABASE`, `DROP DATABASE`
- `CREATE TABLESPACE`, `CLUSTER`, `DISCARD`, `VACUUM`
- ...

`https://wiki.postgresql.org/wiki/Transactional_DDL_in_PostgreSQL:_A_Competitive_Analysis`

Использование транзакций. Субтранзакции

Транзакции можно разделять на субтранзакции.

Внутри транзакции можно сделать:

- `SAVEPOINT <id>` - создает ”точку сохранения” с `<id>`.
- `ROLLBACK TO <id>` - откатывает состояние транзакции до точки `<id>`
- `RELEASE SAVEPOINT <id>` - удаляет ”точку сохранения”.

Не злоупотребляйте - например, не стоит создавать N `SAVEPOINT`-ов в цикле, поскольку каждый следующий SAVEPOINT в одной транзакции будет делаться чуть медленнее.

`https://www.postgresql.org/docs/current/sql-rollback-to.html`

Проблемы конкурентного доступа к данным

Проблемы конкурентного доступа к данным - класс проблем, которые возникают, когда базы данных используют одни и те же данные в параллельных транзакциях.

- lost update - потерянное обновление
- dirty read - ”грязное чтение”
- non-repeatable read - неповторяющееся чтение
- phantom read - фантомное чтение

Потерянное обновление (lost update)

Transaction 1	Transaction 2
`UPDATE tbl SET foo = 2 WHERE id = 1`	
	`UPDATE tbl SET foo = 3 WHERE id = 1`
`COMMIT`	`COMMIT`

Потеряли `foo = 2`

Грязное чтение (dirty read)

Transaction 1	Transaction 2
`UPDATE tbl SET foo = 2 WHERE id = 1`	
	`SELECT foo FROM tbl WHERE id = 1`
`ROLLBACK`	

Выбрали `foo = 2` из незавершенной транзакции.

Неповторяющееся чтение (non-repeatable read)

Transaction 1	Transaction 2
	`SELECT foo FROM tbl WHERE id = 1`
`UPDATE tbl SET foo = 2 WHERE id = 1`	
`COMMIT`	
	`SELECT foo FROM tbl WHERE id = 1`

Сначала выбрали `foo = 1`, потом тем же запросом `foo = 2`.

Фантомное чтение (phantom read)

Transaction 1	Transaction 2
	`SELECT foo FROM tbl WHERE foo > 3`
`INSERT INTO tbl (foo) VALUES (4)`	
`COMMIT`	
	`SELECT foo FROM tbl WHERE foo > 3`

Выбрали разное количество строчек одним и тем же запросом.

Отличие от неповторяющегося чтения в том, что в фантомном чтении выбираются новые строчки (или пропадают старые), а в неповторяющееся чтении получаются разные значения для одних и тех же строчек.

Решение проблем. Блокирующий SELECT

`SELECT` можно выполнить так, чтобы заблокировать другим транзакциям доступ к выбранным строчкам (locking read).

- `SELECT ... FOR SHARE` - строчки можно прочитать в других транзакциях, но не обновить.
- `SELECT ... FOR UPDATE` - строчки нельзя обновить и нельзя сделать `SELECT ... FOR SHARE` (обычный `SELECT` пройдет).

`https://www.postgresql.org/docs/current/explicit-locking.html#LOCKING-`

Решение проблем. Блокирующий SELECT (2)

`SELECT`, упирающийся в блокировку, ”зависнет” до момента отключения блокировки (завершения транзакции, где блокировка установлена). Но можно сделать:

- `SELECT ... FOR <mode> NOWAIT` - вернуть ошибку, если попытались выбрать заблокированные строчки
- `SELECT ... FOR <mode> SKIP LOCKED` - пропустить заблокированные строчки

В обоих случаях запросы выполняются мгновенно.

С помощью блокирующего SELECT можно решить проблемы:

- lost update
- dirty read
- non-repeatable read

заблокировав нужные строчки и фактически пожертвовав параллельностью.

Уровни изоляции

Уровень изоляции - значение компромисса между защитой данных и возможностью параллельного доступа.

```
1  SET TRANSACTION ISOLATION LEVEL <level>;
2  START TRANSACTION;
3  ...
```

Где `<level>`:

- READ UNCOMMITTED - низший уровень защиты
- READ COMMITED
- REPEATABLE READ - значение по умолчанию в PostgreSQL
- SERIALIZABLE
- SNAPSHOT - нет в PostgreSQL (есть в MSSQL)

По умолчанию применятся к следующей транзакции в сессии.

- `SET SESSION TRANSACTION` - применить ко всем транзакциям в сессии
- `SET GLOBAL TRANSACTION` - применить ко всем транзакциям в БД

`https://www.postgresql.org/docs/current/sql-set-transaction.html`

Уровни изоляции. READ UNCOMMITTED

READ UNCOMMITTED - минимальный уровень изоляции.

- Блокировки на чтение не устанавливаются.
- Транзакции могут считывать измененные другими транзакциями, но не закоммиченные строчки.

Поведение UPDATE при READ UNCOMMITTED

Transaction 1	Transaction 2
`SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED`	`SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED`
`START TRANSACTION`	`START TRANSACTION`
`UPDATE tbl SET foo = 2 WHERE id = 1`	
	`UPDATE tbl SET foo = 3 WHERE id = 1`
	UPDATE ”завис”
`COMMIT`	UPDATE прошел

2-й UPDATE ”зависнет”, т.к. упрется в блокировку, повешенную 1-м UPDATE-ом.

Это решает проблему потерянного обновления (но не остальные 3).

В частности, при этом уровне можно считать незакомиченные данные других транзакций (dirty read)

Уровни изоляции. READ COMMITTED

READ COMMITTED - незакомиченные (”грязные”) данные считать нельзя (будет возвращены значения в состоянии БД без транзакций). Это решает проблему dirty read.

Но данные могут быть изменены другими транзакциями: non-repeatable read по-прежнему возможен.

Уровни изоляции. REPEATABLE READ

REPEATABLE READ - нельзя считать данные, которые были закоммичены другими транзакциями.

При 1-м `SELECT` в режиме REPEATABLE READ создается снимок считываемых данных, на который не влияют другие транзакции.

Таким образом, все следующие `SELECT` будут возвращать те же значения (для уже записанных строчек).

Можно запустить транзакцию в режиме:

```
1  START TRANSACTION WITH CONSISTENT SNAPSHOT;
```

Тогда снимок будет создан сразу же.

Это решает проблему non-repeatable read, но не phantom read.

Уровни изоляции. SERIALIZABLE

SERIALIZABLE - REPEATABLE READ + все SELECT-ы становятся FOR SHARE (блокирующими).

Это решает проблему phantom read, но приводит к большому количеству блокировок.

Уровни изоляции. SNAPSHOT

SNAPSHOT - в момент начала транзакции делается снимок базы. Транзакция работает с ним.

Блокировки не устанавливаются. Если в конце транзакции возникает конфликт, транзакция отменяется.

Этого режима нет в PostgreSQL, но есть в MSSQL.

Уровни изоляции. Решение проблем

Уровень изоляции	Lost update	Dirty read	Non-repeatable read	Phantom read
READ UNCOMMITTED	Нет	Да	Да	Да
READ COMMITTED	Нет	Нет	Да	Да
REPEATABLE READ	Нет	Нет	Нет	Да
SERIALIZABLE	Нет	Нет	Нет	Нет
SNAPSHOT	Нет	Нет	Нет	Нет

`https://www.postgresql.org/docs/current/transaction-iso.html`

Решение проблем. Блокировка таблиц

Таблицы можно блокировать на чтение/запись:

- `LOCK TABLES <table-name> READ | WRITE`
- `UNLOCK TABLES`

```
1  LOCK TABLES table_1 READ, table_2 WRITE;
2  <сделать что-то с table_1, table_2>
3  UNLOCK TABLES;
```

Блокировки могут применяться для тех же целей, что и транзакции.

DEADLOCK

При использовании вышеописанных методов есть риск получить DEADLOCK - состояние, в котором две транзакции блокируют друг друга и не могут завершиться.

Чем выше ”размах” блокировок, тем больше шанс.

PostgreSQL обычно распознает deadlock-и, и одна из заблокированных транзакций завершается с ошибкой.

DEADLOCK. Пример

Transaction 1	Transaction 2
`UPDATE tbl1 SET foo = 2 WHERE id = 1`	`UPDATE tbl2 SET bar = 2 WHERE id = 1`
`UPDATE tbl2 SET bar = 3 WHERE id = 1`	
Ждем Transaction 2	
Ждем Transaction 2	`UPDATE tbl1 SET foo = 3 WHERE id = 1`
Ждем Transaction 2	Ждем Transaction 1
T_T	T_T