

Севастопольский государственный университет
Кафедра «Информационные системы»

Управление данными

курс лекций

лектор:
ст. преподаватель кафедры ИС Абрамович А.Ю.



Лекция 7

**Механизм транзакций и блокировки
в базах данных.
Уровни изоляции.
Взаимоблокировки**

КОНЦЕПЦИЯ ТРАНЗАКЦИЙ

Под **транзакцией** понимается неделимая с точки зрения воздействия на БД последовательность операторов манипулирования данными (чтения, удаления, вставки, модификации), приводящая к одному из двух возможных результатов: либо последовательность выполняется, если все операторы правильные, либо вся транзакция прерывается, если хотя бы один оператор не может быть успешно выполнен. Обработка транзакций гарантирует целостность информации в базе данных. **Транзакция переводит базу данных из одного целостного состояния в другое.**



**ПЛОСКИЕ
(КЛАССИЧЕСКИЕ,
ТРАДИЦИОННЫЕ)
ТРАНЗАКЦИИ**

**РАСПРЕДЕЛЕННЫЕ
ТРАНЗАКЦИИ**

**ВЛОЖЕННЫЕ
ТРАНЗАКЦИИ**

Если транзакция состоит из набора операций над объектами, она называется **плоской (flat)**. Основное ограничение плоских транзакций состоит в том, что **клиент не может воспользоваться промежуточными результатами транзакции**, например в случае ее прерывания.

Транзакции, выполняющиеся в составе других транзакций, принято называть **вложенными (nested)**. Вложенные транзакции снимают некоторые ограничения плоских транзакций и могут выполняться параллельно для повышения производительности. **Транзакции, затрагивающие объекты, расположенные на разных машинах, называют распределенными**. Распределенная транзакция может быть как плоской, так и вложенной.

БЛОКИРОВКИ

Повышение эффективности работы при использовании небольших транзакций связано с тем, что при выполнении транзакции сервер накладывает **на данные блокировки**.

Блокировкой называется временное ограничение на выполнение некоторых операций обработки данных. Блокировка может быть наложена как на отдельную строку таблицы, так и на всю базу данных. Управлением блокировками на сервере занимается менеджер блокировок, контролирующий их применение и разрешение конфликтов. **Транзакции накладывают блокировки на данные, чтобы обеспечить выполнение требований ACID.** Без использования блокировок несколько транзакций могли бы изменять одни и те же данные.

Блокировка представляет собой метод управления параллельными процессами, при котором объект БД не может быть модифицирован без ведома транзакции, т.е. происходит блокирование доступа к объекту со стороны других транзакций, чем исключается непредсказуемое изменение объекта.

Различают **два вида** блокировки:

- **блокировка записи** – транзакция блокирует строки в таблицах таким образом, что запрос другой транзакции к этим строкам будет отменен ;
- **блокировка чтения** – транзакция блокирует строки так, что запрос со стороны другой транзакции на блокировку записи этих строк будет отвергнут, а на блокировку чтения – принят.
- **блокировка обновления** – это промежуточная блокировка. Она совместима с блокировкой чтения, но не совместима с блокировкой записи и сама с собой.

Решение проблемы параллельной обработки БД заключается в том, что строки таблиц блокируются, а последующие транзакции, модифицирующие эти строки, отвергаются и переводятся в режим ожидания. В связи со свойством сохранения целостности БД транзакции являются подходящими единицами изолированности пользователей. Если каждый сеанс взаимодействия с базой данных реализуется транзакцией, то пользователь начинает с того, что обращается к согласованному состоянию базы данных – состоянию, в котором она могла бы находиться, даже если бы пользователь работал с ней в одиночку.

В СУБД используют **протокол доступа к данным, позволяющий избежать проблемы параллелизма**. Его суть заключается в следующем:

- **транзакция**, результатом действия которой на строку данных в таблице является ее извлечение, **обязана наложить блокировку чтения на эту строку**;
- **транзакция**, предназначенная для модификации строки данных, **накладывает на нее блокировку записи**;
- если запрашиваемая **блокировка на строку отвергается** из-за уже имеющейся блокировки, то **транзакция переводится в режим ожидания** до тех пор, пока блокировка не будет снята;
- **блокировка записи сохраняется** вплоть до конца выполнения транзакции.

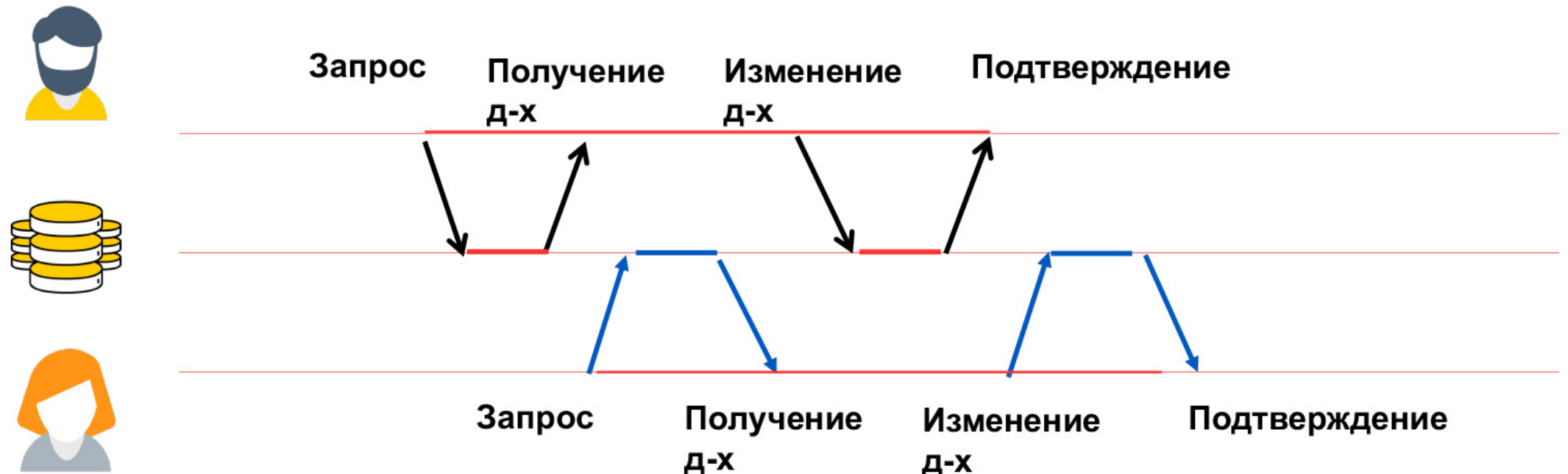
Если в системе управления базами данных не реализованы механизмы блокирования, **то при одновременном чтении и изменении одних и тех же данных несколькими пользователями могут возникнуть следующие проблемы одновременного доступа:**

- **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;
- **«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;
- **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;
- **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;
- **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.

Для решения перечисленных проблем в специально разработанном стандарте определены уровни изоляции.

ПОТЕРЯННОЕ ОБНОВЛЕНИЕ (LOST UPDATE)

При одновременном изменении одного блока данных разными транзакциями одно из изменений теряется. **Две параллельные транзакции меняют одни и те же данные. Итоговый результат обновления предсказать невозможно.**



Если пользователи **параллельно обновляют** одни и те же данные, то **запомненным будет то обновление**, которое **было проведено последним**. Остальные обновления будут потеряны.

ТРАНЗАКЦИЯ 1

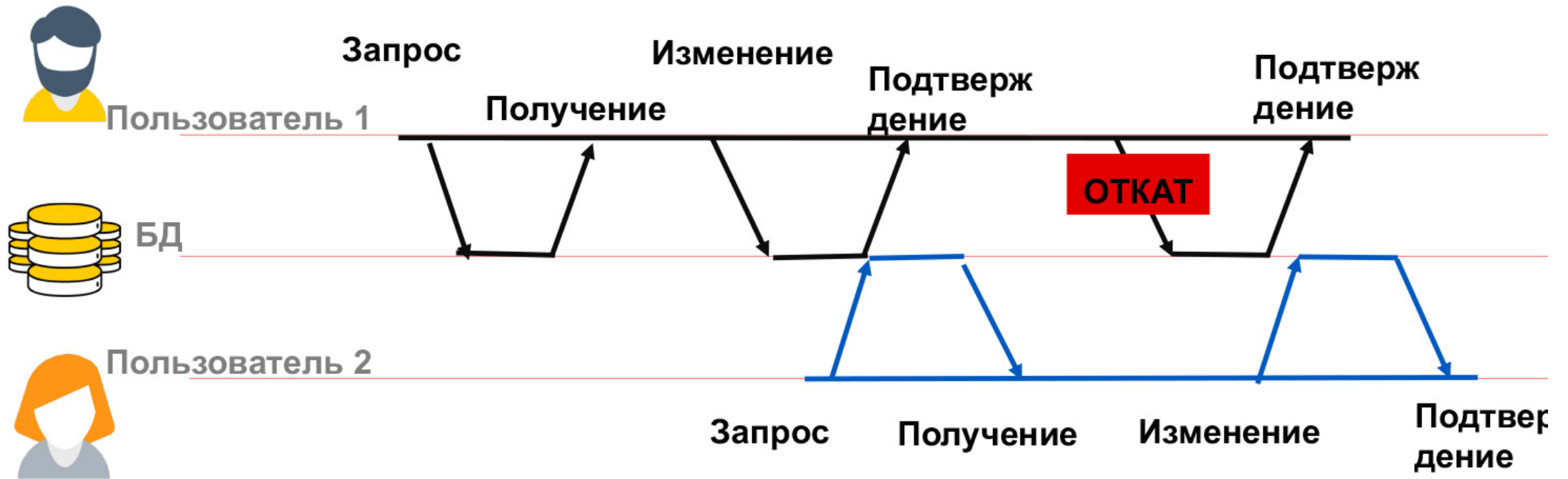
ТРАНЗАКЦИЯ 2

UPDATE tb11 SET f2=f2+20 WHERE f1=1; UPDATE tb11 SET f2=f2+25 WHERE f1=1;

1. Обе транзакции одновременно читают текущее состояние поля.
2. Обе транзакции вычисляют новое значение поля.
3. Транзакции пытаются записать результат вычислений обратно в поле f2. Физически одновременно две записи выполнить невозможно, одна из операций записи будет выполнена раньше, другая позже. При этом вторая операция записи перезапишет результат первой. Первая транзакция «пропадет».

«ГРЯЗНОЕ» ЧТЕНИЕ (DIRTY READ)

Чтение данных, добавленных или изменённых транзакцией, которая впоследствии не подтвердится (откатится). В результатах запроса появляются промежуточные результаты параллельной транзакции, которая ещё не завершилась.



Аномалия грязного чтения (dirty read) возникает, когда **транзакция читает еще не зафиксированные изменения, сделанные другой транзакцией.**

ТРАНЗАКЦИЯ 1

```
UPDATE tbl1 SET f2=f2+1 WHERE f1=1;
```

```
ROLLBACK;
```

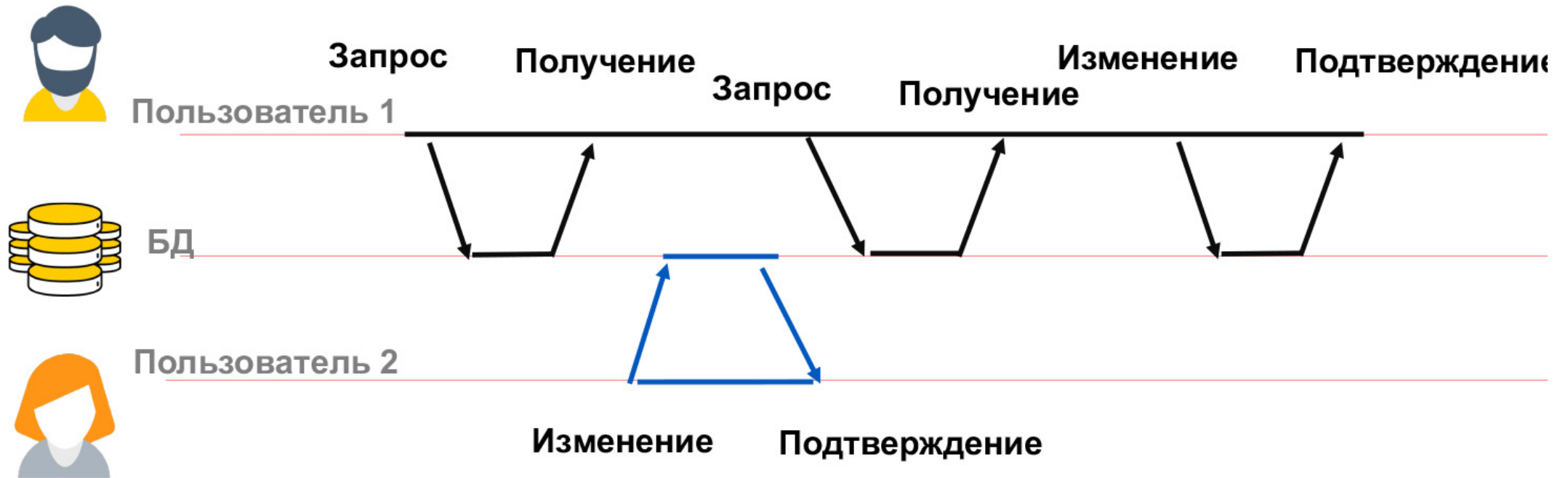
ТРАНЗАКЦИЯ 2

```
SELECT f2 FROM tbl1 WHERE f1=1;
```

В транзакции 1 изменяется значение поля f2, а затем в транзакции 2 выбирается значение этого поля. После этого происходит откат транзакции 1. В результате значение, полученное второй транзакцией, будет отличаться от значения, хранимого в базе данных.

НЕПОВТОРЯЮЩЕЕСЯ ЧТЕНИЕ (NON-REPEATABLE READ)

При повторном чтении в рамках одной транзакции ранее прочитанные данные оказываются изменёнными. **Запрос с одними и теми же условиями даёт неодинаковые результаты в рамках транзакции.**



Аномалия неповторяющегося чтения (nonrepeatable read) возникает, **когда транзакция читает одну и ту же строку два раза, а в промежутке между чтениями вторая транзакция изменяет (или удаляет) эту строку и фиксирует изменения. Тогда первая транзакция получит разные результаты.**

ТРАНЗАКЦИЯ 1

```
UPDATE tbl1 SET f2=f2+1 WHERE f1=1;  
COMMIT;
```

ТРАНЗАКЦИЯ 2

```
SELECT f2 FROM tbl1 WHERE f1=1;
```

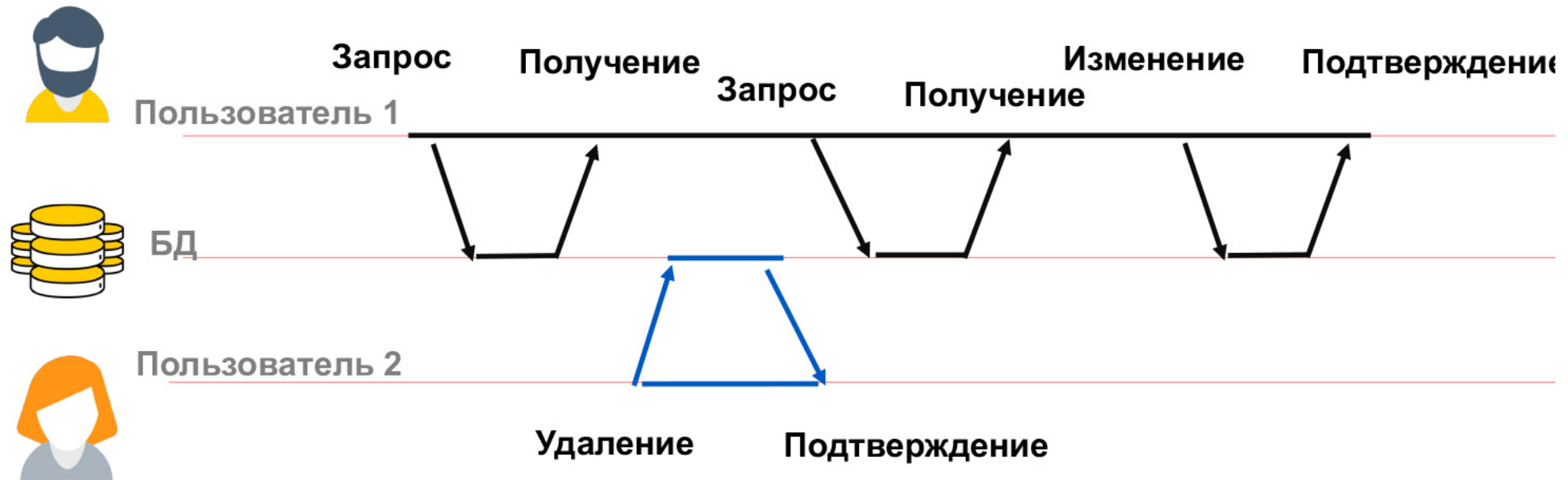
```
SELECT f2 FROM tbl1 WHERE f1=1;
```

В транзакции 2 выбирается значение поля f2, затем в транзакции 1 изменяется значение поля f2. При повторной попытке выбора значения из поля f2 в транзакции 2 будет получен другой результат.

Эта ситуация особенно неприемлема, когда данные считываются с целью их частичного изменения и обратной записи в базу данных.

ФАНТОМНОЕ ЧТЕНИЕ (PHANTOM READ)

При повторном чтении в рамках одной транзакции одна и та же выборка дает разные множества строк. В результатах повторяющегося запроса появляются и исчезают строки, которые модифицирует параллельная транзакция.



Аномалия фантомного чтения (phantom read) возникает, когда **одна транзакция 2 раза читает набор строк по одинаковому условию, а в промежутке между чтениями другая транзакция добавляет строки, удовлетворяющие этому условию, и фиксирует изменения.** Тогда первая транзакция получит разные наборы строк.

ТРАНЗАКЦИЯ 1

```
INSERT INTO tb11 (f1,f2) VALUES (15,20);  
COMMIT;
```

ТРАНЗАКЦИЯ 2

```
SELECT SUM(f2) FROM tb11;
```

```
SELECT SUM(f2) FROM tb11;
```

В транзакции 2 выполняется SQL-оператор, использующий все значения поля f2. Затем в транзакции 1 выполняется вставка новой строки, приводящая к тому, что повторное выполнение SQL-оператора в транзакции 2 выдаст другой результат.

АНОМАЛИИ СЕРИАЛИЗАЦИИ

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.

Аномалии сериализации – ситуация, когда параллельное выполнение транзакций приводит к результату, невозможному при последовательном выполнении тех же транзакций.

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.

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

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

READ UNCOMMITTED

1

READ COMMITTED

2

REPEATABLE READ

3

SERIALIZABLE

4

Каждый более высокий уровень включает в себя все возможности предыдущего.

		ЭФФЕКТЫ				
		ПОТЕРЯННОЕ ОБНОВЛЕНИЕ	ГРЯЗНОЕ ЧТЕНИЕ	НЕПОВТОР-СЯ ЧТЕНИЕ	ФАНТОМНОЕ ЧТЕНИЕ	АНОМАЛИИ СЕРИАЛИЗАЦИИ
УРОВНИ ИЗОЛЯЦИИ	READ UNCOMMITTED	Нет	Допускается	Возможно	Возможно	Возможно
	READ COMMITTED	Нет	Нет	Возможно	Возможно	Возможно
	REPEATABLE READ	Нет	Нет	Нет	Допускается	Возможно
	SERIALIZABLE	Нет	Нет	Нет	Нет	Нет

Повышение уровня изоляции: точность и согласованность данных растет, количество параллельно выполняемых транзакций уменьшается.

READ UNCOMMITTED – ЧТЕНИЕ НЕЗАФИКСИРОВАННЫХ ДАННЫХ

Гарантирует **только отсутствие** потерянных обновлений. Если несколько параллельных транзакций пытаются изменять одну и ту же строку таблицы, то в **окончательном варианте строка будет иметь значение, определенное всем набором успешно выполненных транзакций.**

При этом **возможно считывание** не только логически несогласованных данных, но и данных, изменения которых ещё не зафиксированы.

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.**

В PostgreSQL можно запросить любой из четырех стандартных уровней изоляции транзакций, но внутренне реализованы только три различных уровня изоляции, т. е. **режим PostgreSQL Read Uncommitted ведет себя как Read Committed.** Это потому, что это **единственный разумный способ сопоставить стандартные уровни изоляции с многоверсионной архитектурой управления параллелизмом PostgreSQL.**

READ COMMITTED – ЧТЕНИЕ ЗАФИКСИРОВАННЫХ ДАННЫХ

Если транзакция начала изменение данных, то никакая другая транзакция не сможет прочитать их до завершения первой.

В процессе работы одной транзакции другая может быть успешно завершена и сделанные ею изменения зафиксированы. В итоге первая транзакция будет работать с другим набором данных.

Этот уровень изоляции **используется по умолчанию в большинстве реляционных СУБД**, в том числе и в **PostgreSQL**, и в **Oracle**, и в др. Он гарантирует, что никогда не будут прочитаны «грязные» данные. То есть **другая транзакция никогда не видит промежуточных этапов первой транзакции**. Гарантируется, что никогда не будет ситуации, когда видны какие-то части данных, недописанные данные.

От чего не защищает этот уровень изоляции? Он не защищает от того, что **данные, которые участвовали в запросе, могут быть изменены**. В случае небольших запросов этого уровня изоляции вполне достаточно, но для больших, долгих запросов, сложной аналитики, можно использовать более сложные уровни, которые блокируют таблицы.

```
maxmobiles.ru — psql -p5432 sbrmvch — 80x24
Last login: Thu Nov 30 23:55:39 on ttys000
/Applications/Postgres.app/Contents/Versions/11/bin/psql -p5432 "sbrmvch"
sbrmvch:~ sbrmvch$ /Applications/Postgres.app/Contents/Versions/11/bin/psql -p5432 "sbrmvch"
psql (11.19)
Type "help" for help.

[sbrmvch=# BEGIN ISOLATION LEVEL READ COMMITTED;
BEGIN
[sbrmvch=# SHOW transaction_isolation;
transaction_isolation
-----
read committed
(1 row)

[sbrmvch=# update purchases set cost=cost+1000.00 where id=13;
UPDATE 1
[sbrmvch=# select * from purchases where id=13;
id | name | cost | user_id
---+-----+-----+-----
13 | ACER | 1800.00 | 7
(1 row)
```

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

Транзакция видит незафиксированные изменения, выполненные в ней самой

```
maxmobiles.ru — psql -p5432 sbrmvch — 80x24
Last login: Thu Nov 30 23:55:39 on ttys000
/Applications/Postgres.app/Contents/Versions/11/bin/psql -p5432 "sbrmvch"
sbrmvch:~ sbrmvch$ /Applications/Postgres.app/Contents/Versions/11/bin/psql -p5432 "sbrmvch"
psql (11.19)
Type "help" for help.

[sbrmvch=# BEGIN ISOLATION LEVEL READ COMMITTED;
BEGIN
[sbrmvch=# SHOW transaction_isolation;
transaction_isolation
-----
read committed
(1 row)

[sbrmvch=# update purchases set cost=cost+1000.00 where id=13;
UPDATE 1
[sbrmvch=# select * from purchases where id=13;
 id | name | cost  | user_id
----+-----+-----+-----
 13 | ACER | 1800.00 |      7
(1 row)

maxmobiles.ru — psql -p5432 sbrmvch — 80x5
Invalid command \t;. Try \? for help.
[sbrmvch=# begin;
BEGIN
[sbrmvch=# update purchases set cost=cost+3000.00 where id=13;
```

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

```
maxmobiles.ru — psql -p5432 sbrmvch — 80x24
sbrmvch:~ sbrmvch$ /Applications/Postgres.app/Contents/Versions/11/bin/psql -p54
32 "sbrmvch"
psql (11.19)
Type "help" for help.

[sbrmvch=# BEGIN ISOLATION LEVEL READ COMMITTED;
BEGIN
[sbrmvch=# SHOW transaction_isolation;
transaction_isolation
-----
read committed
(1 row)

[sbrmvch=# update purchases set cost=cost+1000.00 where id=13;
UPDATE 1
[sbrmvch=# select * from purchases where id=13;
 id | name | cost  | user_id
-----+-----+-----+-----
 13 | ACER | 1800.00 |      7
(1 row)

[sbrmvch=# commit;
COMMIT
sbrmvch=# ]

maxmobiles.ru — psql -p5432 sbrmvch — 80x5
[sbrmvch=# begin;
BEGIN
[sbrmvch=# update purchases set cost=cost+3000.00 where id=13;
UPDATE 1
sbrmvch=# ]
```

```

maxmobiles.ru — psql -p5432 sbrmvch — 80x24
sbrmvch:~ sbrmvch$ /Applications/Postgres.app/Contents/Versions/11/bin/psql -p54
32 "sbrmvch"
psql (11.19)
Type "help" for help.

[sbrmvch=# BEGIN ISOLATION LEVEL READ COMMITTED;
BEGIN
[sbrmvch=# SHOW transaction_isolation;
transaction_isolation
-----
read committed
(1 row)

[sbrmvch=# update purchases set cost=cost+1000.00 where id=13;
UPDATE 1
[sbrmvch=# select * from purchases where id=13;
 id | name | cost  | user_id
-----+-----+-----+-----
 13 | ACER | 1800.00 |      7
(1 row)

[sbrmvch=# commit;
COMMIT

```

```

maxmobiles.ru — psql -p5432 sbrmvch — 80x12
BEGIN
[sbrmvch=# update purchases set cost=cost+3000.00 where id=13;
UPDATE 1
[sbrmvch=# select * from purchases where id=13;
 id | name | cost  | user_id
-----+-----+-----+-----
 13 | ACER | 4800.00 |      7
(1 row)

[sbrmvch=# end;
COMMIT

```

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.

REPEATABLE READ (SNAPSHOT ISOLATION) – ПОВТОРЯЕМОСТЬ ЧТЕНИЯ

Уровень изоляции Repeatable read защищает от **первых трех проблем**: потерянное обновление, «грязное» чтение и неповторяющееся чтение.

Уровень изоляции Repeatable Read видит только данные, зафиксированные до начала транзакции; он никогда не видит ни незафиксированные данные, ни изменения, зафиксированные во время выполнения транзакции параллельными транзакциями.

Если транзакция считывает данные, то никакая другая транзакция не сможет их изменить. При повторном чтении они будут находиться в первоначальном состоянии.

Но другие транзакции могут вставлять новые строки, соответствующие условиям поиска инструкций, содержащихся в текущей транзакции. При повторном запуске инструкции текущей транзакцией будут извлечены новые строки, что приведёт к **фантомному чтению**.

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.

```
maxmobiles.ru — psql p5432 sbrmvch — 80x24

(12 rows)

[sbrmvch=# end;
COMMIT
[sbrmvch=# BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN
[sbrmvch=# select * from purchases;
id |          name          | cost   | user_id
---+-----+-----+-----
 1 | M1 MacBook Air        | 1300.99 |      1
 2 | Iphone 14             | 1200.00 |      2
 3 | Iphon 10              |  700.00 |      3
 4 | Iphone 13             |  800.00 |      1
 5 | Intel Core i5         |  500.00 |      4
 7 | IMAC                  | 2500.00 |      7
 8 | ASUS VIVOBBOOK        |  899.99 |      6
10 | Galaxy S21            |  999.99 |      2
12 | M1 MacBook Air        | 1299.99 |      8
 6 | M1 MacBook Pro        |  800.00 |      5
11 | XIAMI REDMIBOOK 14    |  800.00 |      4
13 | ACER                  | 4800.00 |      7
(12 rows)

sbrmvch=#
```

```
maxmobiles.ru — psql -p5432 sbrmvch — 80x12

[sbrmvch=# end;
COMMIT
[sbrmvch=# BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN
[sbrmvch=# insert into purchases values (14, 'iphone 15', 9999.99, 2);
INSERT 0 1
[sbrmvch=# update purchases set cost=cost+3000.00 where id=13;
UPDATE 1
[sbrmvch=# end
[sbrmvch=# ;
COMMIT
sbrmvch=#
```

BEGIN

[sbrmvch=# select * from purchases;

id	name	cost	user_id
1	M1 MacBook Air	1300.99	1
2	Iphone 14	1200.00	2
3	Iphon 10	700.00	3
4	Iphone 13	800.00	1
5	Intel Core i5	500.00	4
7	IMAC	2500.00	7
8	ASUS VIVOB00K	899.99	6
10	Galaxy S21	999.99	2
12	M1 MacBook Air	1299.99	8
6	M1 MacBook Pro	800.00	5
11	XIAMI REDMIBOOK 14	800.00	4
13	ACER	4800.00	7

(12 rows)

[sbrmvch=# select * from purchases;

id	name	cost	user_id
1	M1 MacBook Air	1300.99	1
2	Iphone 14	1200.00	2
3	Iphon 10	700.00	3
4	Iphone 13	800.00	1
5	Intel Core i5	500.00	4
7	IMAC	2500.00	7
8	ASUS VIVOB00K	899.99	6
10	Galaxy S21	999.99	2
12	M1 MacBook Air	1299.99	8
6	M1 MacBook Pro	800.00	5
11	XIAMI REDMIBOOK 14	800.00	4
13	ACER	4800.00	7

(12 rows)

[sbrmvch=# end;

COMMIT

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.

```
maxmobiles.ru — psql -p5432 sbrmvch — 80x21
[sbrmvch=# end;
COMMIT
[sbrmvch=# select * from purchases;
 id |      name      |  cost  | user_id
----+-----+-----+-----
  1 | M1 MacBook Air | 1300.99 |      1
  2 | Iphone 14      | 1200.00 |      2
  3 | Iphon 10       |  700.00 |      3
  4 | Iphone 13      |  800.00 |      1
  5 | Intel Core i5  |  500.00 |      4
  7 | IMAC           | 2500.00 |      7
  8 | ASUS VIVOBOKK  |  899.99 |      6
 10 | Galaxy S21     |  999.99 |      2
 12 | M1 MacBook Air | 1299.99 |      8
  6 | M1 MacBook Pro |  800.00 |      5
 11 | XIAMI REDMIBOOK 14 | 800.00 |      4
 14 | iphone 15      | 9999.99 |      2
 13 | ACER           | 7800.00 |      7
(13 rows)

sbrmvch=#
```

Но до тех пор, пока на первом терминале транзакция находилась в процессе выполнения первой, все эти изменения не были ей доступны, поскольку первая транзакция использовала снимок, сделанный до внесения изменений и их фиксации второй транзакцией.

SERIALIZABLE – УПОРЯДОЧИВАЕМОСТЬ (БЛОКИРУЕТ ЧТЕНИЕ)

Если транзакция обращается к данным, то никакая другая транзакция не сможет добавить новые или удалить имеющиеся строки, которые могут быть считаны при выполнении транзакции.

Такая блокировка накладывается **не на конкретные строки таблицы, а на строки, удовлетворяющие определенному логическому условию.**

Уровень изоляции **Serializable** обеспечивает самую строгую изоляцию транзакций. Этот уровень эмулирует последовательное выполнение транзакций для всех зафиксированных транзакций; как если бы транзакции выполнялись одна за другой, последовательно, а не одновременно.

Как и уровень **Repeatable Read**, приложения, использующие этот уровень, должны быть готовы к повторным попыткам транзакций из-за сбоев сериализации.

Фактически, этот уровень изоляции работает точно так же, как **Repeatable Read**, за исключением того, что он также отслеживает условия, которые могут привести к тому, что выполнение параллельного набора сериализуемых транзакций будет вести себя несовместимо со всеми возможными последовательными (по одному) выполнениями этих транзакций.

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

По умолчанию PostgreSQL использует уровень изоляции **READ COMMITTED**.

```
SHOW default_transaction_isolation;
```

```
test1=# SHOW default_transaction_isolation;
 default_transaction_isolation
-----
 read committed
(1 row)
```

ВЗАИМОБЛОКИРОВКИ (DEADLOCKS)

Ситуация, при которой **две транзакции блокируют друг друга**, и ни одна из них **не может продолжать** свое выполнение.

ПРИМЕР ВЗАИМОБЛОКИРОВКИ

СТАРТ ТРАНЗАКЦИИ	ТРАНЗАКЦИЯ 1	ТРАНЗАКЦИЯ 2
ШАГ 1	Накладывает совмещаемую блокировку на ресурс 1	Накладывает совмещаемую блокировку на ресурс 2
ШАГ 2	Запрашивает эксклюзивную блокировку на ресурс 2	Запрашивает эксклюзивную блокировку на ресурс 1
ШАГ 3	Ожидает, когда с ресурса 2 будет снята совмещаемая блокировка , наложенная Транзакцией 2 на первом шаге	Ожидает, когда с ресурса 1 будет снята совмещаемая блокировка , наложенная Транзакцией 1 на первом шаге
ШАГ 4	ВЗАИМОБЛОКИРОВКА (DEADLOCKS)	

Если возникла **взаимоблокировка**, то обе транзакции **будут заблокированы вечно**, т.е. ни одна из них не будет выполнена, поэтому данная ситуация, т.е. Deadlock, требует вмешательства извне.

Классический пример, поясняющий суть транзакции базы данных, — это **банковский перевод со счёта на счёт**. Предположим, приложение предоставляет возможность выполнить перевод какой-то суммы со счёта А на счёт Б.

```
sbrmvch=# create table account (acct_id integer, amount integer);
```

```
[CREATE TABLE  
sbrmvch=# insert into account values (1, 500);
```

```
[INSERT 0 1  
sbrmvch=# insert into account values (2, 300);
```

```
[INSERT 0 1  
sbrmvch=# select * from account;
```

[acct_id amount	
-----+-----	
1	500
2	300
(2 rows)	

При осуществлении перевода эта сумма должна быть списана со счёта А и зачислена на счёт Б. Списание и зачисление образуют единую логическую единицу работы. Должны выполняться обе операции, в противном случае не будет выполнена ни одна из них. **Вот почему два этих действия должны быть частью одной транзакции.**

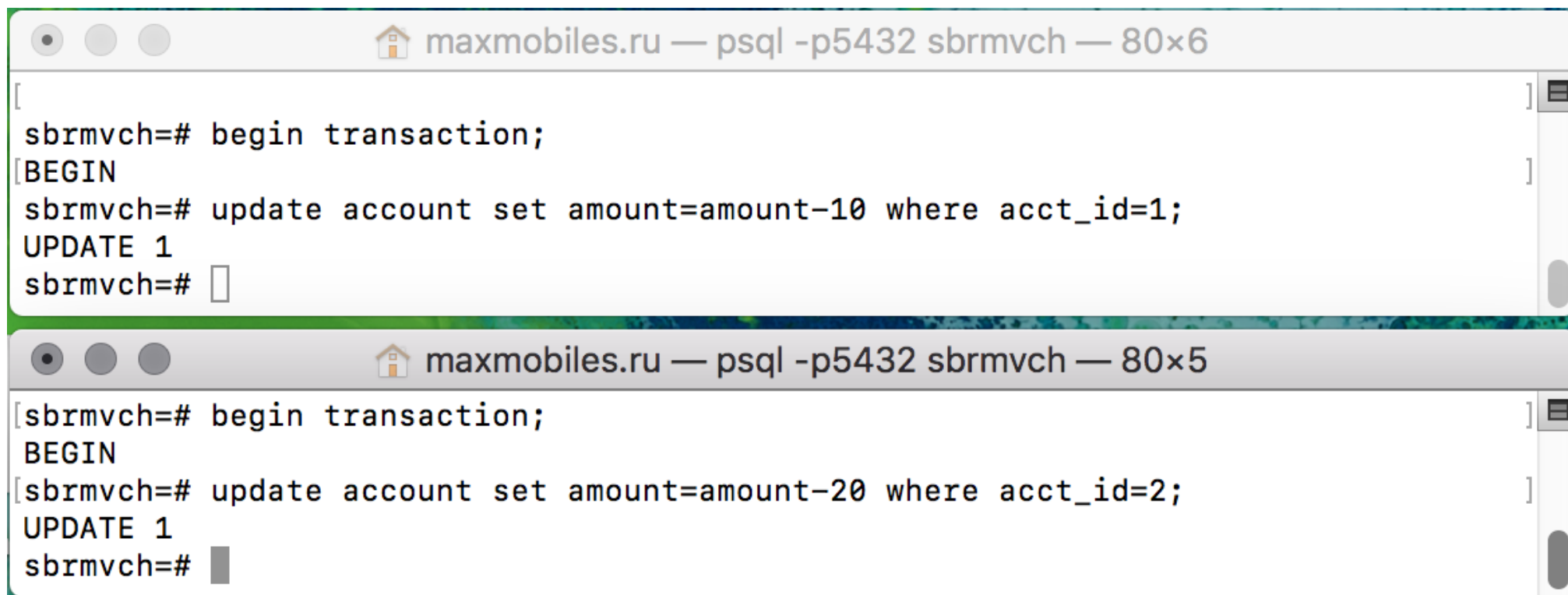
```
sbrmvch=# begin transaction;  
[BEGIN  
sbrmvch=# update account set amount=amount-10 where acct_id=1;  
[UPDATE 1  
sbrmvch=# update account set amount=amount+10 where acct_id=2;  
[UPDATE 1  
sbrmvch=# commit;
```

```
[COMMIT  
sbrmvch=# select * from account;
```

[acct_id amount	
-----+-----	
1	490
2	310
(2 rows)	

ИСКУССТВЕННО СОЗДАДИМ СИТУАЦИЮ ВЗАИМОБЛОКИРОВКИ

Любая СУБД в состоянии эксплуатационной готовности способна обслуживать несколько **одновременных процессов**. Смоделируем два денежных перевода, осуществляемых одновременно. Для этого запустим две оболочки psql.

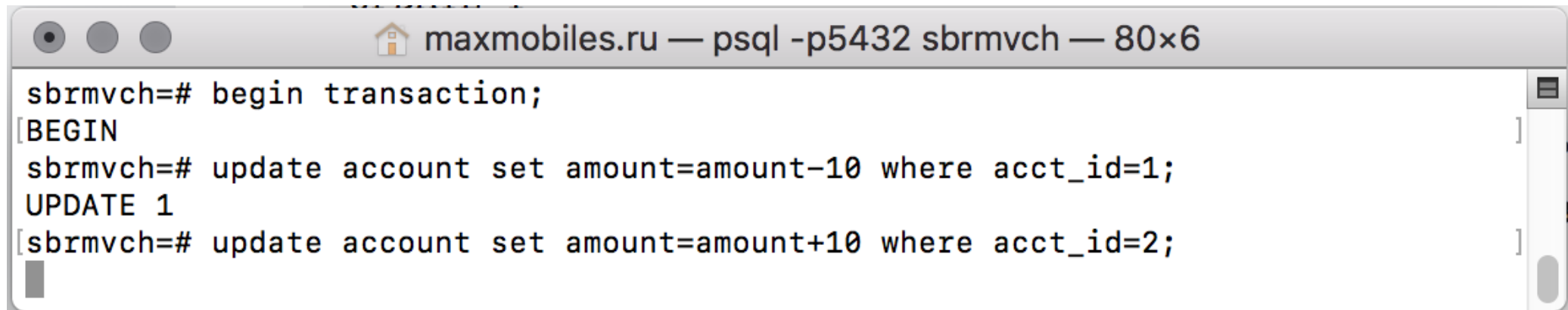


The image shows two terminal windows side-by-side, both titled 'maxmobiles.ru — psql -p5432 sbrmvch — 80x6' and 'maxmobiles.ru — psql -p5432 sbrmvch — 80x5' respectively. The top window (80x6) contains the following commands: `[sbrmvch=# begin transaction;`, `[BEGIN`, `sbrmvch=# update account set amount=amount-10 where acct_id=1;`, `UPDATE 1`, and `sbrmvch=#` with a cursor. The bottom window (80x5) contains: `[sbrmvch=# begin transaction;`, `BEGIN`, `[sbrmvch=# update account set amount=amount-20 where acct_id=2;`, `UPDATE 1`, and `sbrmvch=#` with a cursor.

```
[ sbrmvch=# begin transaction;
[BEGIN
sbrmvch=# update account set amount=amount-10 where acct_id=1;
UPDATE 1
sbrmvch=#
```

```
[sbrmvch=# begin transaction;
BEGIN
[sbrmvch=# update account set amount=amount-20 where acct_id=2;
UPDATE 1
sbrmvch=#
```

В первой оболочке будет имитироваться процесс, в ходе которого выполняется перевод суммы со счёта 1 на счёт 2. Во второй оболочке будет имитироваться процесс, в ходе которого выполняется перевод суммы со счёта 2 на счёт 1.



```
maxmobiles.ru — psql -p5432 sbrmvch — 80x6
sbrmvch=# begin transaction;
[BEGIN
sbrmvch=# update account set amount=amount-10 where acct_id=1;
UPDATE 1
[sbrmvch=# update account set amount=amount+10 where acct_id=2;
```

Но база данных **не вернёт в ответ сообщение об успешном выполнении**, вместо этого **она заблокируется**, и не будет получено никакого ответа.

Это произойдёт потому, что строка с **acct_id=2** в этот момент **окажется заблокированной** из-за того, что процесс 2 производил обновление этой строки. Процесс 1 не может захватить эту блокировку, пока процесс 2 не освободит её.

```
maxmobiles.ru — psql -p5432 sbrmvch — 80x12
[sbrmvch=# begin transaction;
BEGIN
[sbrmvch=# update account set amount=amount-20 where acct_id=2;
UPDATE 1
[sbrmvch=# update account set amount=amount+20 where acct_id=1;
ERROR:  deadlock detected
DETAIL:  Process 94639 waits for ShareLock on transaction 658; blocked by process 94483.
Process 94483 waits for ShareLock on transaction 659; blocked by process 94639.
HINT:  See server log for query details.
CONTEXT:  while updating tuple (0,3) in relation "account"
```

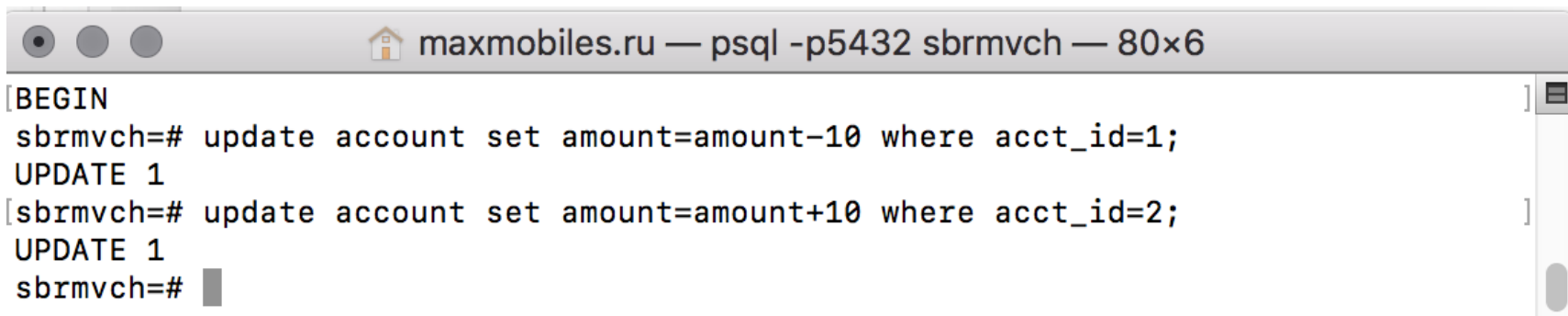
База данных вернёт ERROR: **deadlock detected (ОШИБКА: обнаружена взаимоблокировка).**

Строка базы данных, содержащая `acct_id=1`, была заблокирована процессом 1 при списании суммы 10 со счёта 1. Далее процесс 2 попытался произвести обновление в этой же строке, а для этого ему нужна блокировка. Процесс 2 может захватить блокировку только в том случае, если процесс 1 освободит её. Но процесс 1 блокируется в ожидании, когда процесс 2 освободит блокировку строки `acct_id=2`. Получается, что процесс 1 ожидает освобождения блокировки, захваченной процессом 2, а процесс 2 ожидает освобождения блокировки, захваченной процессом 1. Это и есть взаимоблокировка.

Для базы данных не составляет труда обнаружить взаимную блокировку.

В этом случае в базе данных выдаётся ошибка взаимоблокировки **в оболочке процесса 2**. После возникновения ошибки взаимоблокировки все блокировки, захваченные этим процессом, освобождаются, а **у процесса 1 появляется возможность захватить необходимую ему блокировку**.

Что происходит тогда в оболочке процесса 1: **заблокированная команда возвращается, а сумма 10 зачисляется на счёт 2:**



```
maxmobiles.ru — psql -p5432 sbrmvch — 80x6
[BEGIN
sbrmvch=# update account set amount=amount-10 where acct_id=1;
UPDATE 1
[sbrmvch=# update account set amount=amount+10 where acct_id=2;
UPDATE 1
sbrmvch=#
```

```
maxmobiles.ru — psql -p5432 sbrmvch — 80x12
[sbrmvch=# update account set amount=amount-20 where acct_id=2;
UPDATE 1
[sbrmvch=# update account set amount=amount+20 where acct_id=1;
ERROR:  deadlock detected
DETAIL:  Process 94639 waits for ShareLock on transaction 658; blocked by proces
s 94483.
Process 94483 waits for ShareLock on transaction 659; blocked by process 94639.
HINT:  See server log for query details.
CONTEXT:  while updating tuple (0,3) in relation "account"
[sbrmvch=# commit;
ROLLBACK
[sbrmvch=# ]

maxmobiles.ru — psql -p5432 sbrmvch — 80x6
UPDATE 1
[sbrmvch=# update account set amount=amount+10 where acct_id=2;
UPDATE 1
[sbrmvch=# commit;
COMMIT
[sbrmvch=# ]
```

Устранение и предотвращение взаимоблокировок должно осуществляться на уровне приложения. С этой целью используется код обработки исключений: он отлавливает ошибки взаимоблокировки, давая возможность повторно выполнить неудавшуюся транзакцию.

```
[sbrmvch=# select * from account;
 acct_id | amount 
-----+-----
         1 |      480
         2 |      320
(2 rows)
```