Практические работы по дисциплине «Технологии обработки транзакций клиент-серверных приложений» направления подготовки бакалавриата 09.03.04 «Программная инженерия»

Практическая работа №2

Транзакции. Уровни изоляции. Уровни изоляции в PostgreSQL Read Committed, Repeatable Read. Точки сохранения.

Теория для понимания практики, желательна к выполнению:

Создаем тестовую БД для объяснения базового использования транзакций.

Давайте создадим таблицу дисциплин, читаемых в вузе:

```
test=# CREATE TABLE courses(
test(# c_no text PRIMARY KEY,
test(# title text,
test(# hours integer
test(# );
CREATE TABLE
```

Добавим в созданную таблицу несколько строк:

```
test=# INSERT INTO courses(c_no, title, hours)
VALUES ('CS301', 'Базы данных', 60),
('CS305', 'Сети ЭВМ', 60);
INSERT 0 2
```

Для дальнейших примеров нам потребуется еще две таблицы: студенты и экзамены. Для каждого студента будем хранить его имя и год поступления; идентифицироваться он будет числовым номером студенческого билета.

```
test=# CREATE TABLE students(
s_id integer PRIMARY KEY,
name text,
start_year integer
);
CREATE TABLE
```

```
test=# INSERT INTO students(s_id, name, start_year)
VALUES (1451, 'Анна', 2014),
(1432, 'Виктор', 2014),
(1556, 'Нина', 2015);
INSERT 0 3
```

Запись в таблице экзаменов идентифицируется совокупностью номера студбилета и номера курса. Такое ограничение целостности, относящее сразу к нескольким столбцам, определяется с помощью фразы CONSTRAINT:

```
test=# CREATE TABLE exams(
s_id integer REFERENCES students(s_id),
c_no text REFERENCES courses(c_no),
score integer,
CONSTRAINT pk PRIMARY KEY(s_id, c_no)
);
CREATE TABLE
```

Поставим нашим студентам несколько оценок:

```
test=# INSERT INTO exams(s_id, c_no, score)
VALUES (1451, 'CS301', 5),
(1556, 'CS301', 5),
(1451, 'CS305', 5);
INSERT 0 4
```

Давайте немного расширим нашу схему данных и распределим студентов по группам. При этом потребуем, чтобы у каждой группы в обязательном порядке был староста. Для этого создадим таблицу групп:

```
test=# CREATE TABLE groups( g_no text PRIMARY KEY,
monitor integer NOT NULL REFERENCES students(s_id)
);
CREATE TABLE
```

Здесь мы использовали ограничение целостности NOT NULL, которое

запрещает неопределенные значения.

Теперь в таблице студентов нам необходим еще один столбец — номер группы, — о котором мы не подумали сразу. К счастью, в уже существующую таблицу можно добавить новый столбец:

```
test=# ALTER TABLE students
ADD g_no text REFERENCES groups(g_no);
ALTER TABLE
```

С помощью команды psql всегда можно посмотреть, какие столбцы определены в таблице:

```
test=# \d students
Table "public.students" Column    |  Type  |
Modifiers
-----------+---------+----------
s_id| integer | not null
name | text | start_year | integer | g_no  | text |
```

Также можно вспомнить, какие вообще таблицы присутствуют в базе данных:

```
test=# \d

List of relations
Schema | Name| Type  | Owner
--------+----------+-------+----------
public | courses | table | postgres public | exams |
table | postgres public | groups | table | postgres
public | students | table | postgres
(4 rows)
```

Создадим теперь группу «А-101» и поместим в нее всех студентов, а старостой сделаем Анну.

Тут возникает затруднение. С одной стороны, мы не можем создать группу, не указав старосту. А с другой, как мы можем назначить Анну старостой, если она еще не входит в группу? Это привело бы к появлению в базе данных логически некорректных, несогласованных данных.

Мы столкнулись с тем, что две операции надо совершить одновременно, потому что ни одна из них не имеет смысла без другой. Такие операции, составляющие логически неделимую единицу работы, называются **транзакцией**.

Начнем транзакцию:

```
test=# BEGIN;

BEGIN
```

Затем добавим группу вместе со старостой. Поскольку мы не помним наизусть номер студенческого билета Анны, выполним запрос прямо в команде добавления строк:

```
test=*# INSERT INTO groups(g_no, monitor)  59
SELECT 'A-101', s_id  iv
FROM students
WHERE name = 'Анна';
INSERT 0 1
```

«Звездочка» в приглашении напоминает о незавершенной транзакции.

Откройте теперь новое окно терминала и запустите еще один процесс psql: это будет сеанс, работающий параллельно с первым. Чтобы не запутаться, команды второго сеанса мы будем показывать с отступом.

Увидит ли второй сеанс сделанные изменения?

```
postgres=# \c test
You are now connected to database "test" as user "postgres".
test=# SELECT * FROM groups;
g_no | monitor
------+---------
(0 rows)
```

Нет, не увидит, ведь транзакция еще не завершена. Теперь переведем всех студентов в созданную группу:

```
test=*# UPDATE students SET g_no = 'A-101';
UPDATE 3
```

И снова второй сеанс видит согласованные данные, актуальные на начало еще не оконченной транзакции:

```
test=# SELECT * FROM students;
iv  s_id |  name  | start_year | g_no
------+--------+------------+------
1451 | Анна   |    2014 |
1432 | Виктор |   2014 |
1556 | Нина   |    2015 |
(3 rows)
```

А теперь завершим транзакцию, зафиксировав все сделанные изменения:

```
test=*# COMMIT;
COMMIT
```

И только в этот момент второму сеансу становятся доступны все

изменения, сделанные в транзакции, как будто они появились одномоментно:

```
test=# SELECT * FROM groups;
g_no | monitor
-------+---------
A-101 |  1451
(1 row)

test=# SELECT * FROM students;
s_id |  name  | start_year | g_no
------+--------+------------+------1451 | Анна |
2014 | A-101
1432 | Виктор |  2014 | A-101
1556 | Нина  |   2015 | A-101 (3 rows)
```

СУБД дает несколько очень важных гарантий.

Во-первых, любая транзакция либо выполняется целиком (как в нашем примере), либо не выполняется совсем. Если бы в одной из команд произошла ошибка, или мы сами

прервали бы транзакцию командой ROLLBACK, то база данных осталась бы в том состоянии, в котором она была до команды BEGIN. Это свойство называется **атомарностью**.

Во-вторых, когда фиксируются изменения транзакции, все ограничения целостности должны быть выполнены, иначе транзакция прерывается. В начале работы транзакции данные находятся в согласованном состоянии, и в конце своей работы транзакция оставляет их согласованными; это свойство так и называется — **согласованность**.

В-третьих, как мы убедились на примере, другие пользователи никогда не увидят несогласованные данные, которые транзакция еще не зафиксировала.

Это свойство называется **изоляцией**; за счет его соблюдения СУБД способна параллельно обслуживать много сеансов, не жертвуя корректностью данных. Особенностью PostgreSQL является очень эффективная реализация изоляции: несколько сеансов могут одновременно читать и изменять данные, не блокируя друг друга. Блокировка возникает только при одновременном изменении одной и той же строки двумя разными процессами.

И в-четвертых, гарантируется **долговечность**: зафиксированные данные не пропадут даже в случае сбоя (конечно, при правильных настройках и регулярном выполнении резервного копирования).

Это крайне полезные свойства, без которых невозможно представить себе реляционную систему управления базами данных.

Согласованность

Важная особенность реляционных СУБД — обеспечение *согласованности* (consistency), то есть *корректности* данных.

Известно, что на уровне базы данных можно создавать *ограничения целостности* (integrity constraints), такие как NOT NULL или UNIQUE. СУБД следит за тем, чтобы данные никогда не нарушали эти ограничения, то есть оставались целостными.

Если бы все ограничения были сформулированы на уровне базы данных, согласованность была бы гарантирована. Но некоторые условия слишком сложны для этого, например охватывают сразу несколько таблиц. И даже если ограничение в принципе можно было бы определить в базе данных, но из каких-то соображений оно не определено, это не означает, что его можно нарушать.

Итак, согласованность строже, чем целостность, но что конкретно под ней понимается, СУБД не знает. Если приложение нарушит согласованность, не нарушая целостности, у СУБД не будет способа узнать об этом. Получается,

что гарантом согласованности выступает приложение, и остается верить, что оно написано корректно и никогда не ошибается.

Но если приложение выполняет только корректные последовательности операторов, в чем тогда роль СУБД?

Во-первых, корректная последовательность операторов может временно нарушать согласованность данных, и это, как ни странно, нормально.

Заезженный, но понятный пример состоит в переводе средств с одного счета на другой. Правило согласованности может звучат так: *перевод никогда не меняет общей суммы денег на счетах*. Такое правило довольно трудно (хотя и возможно) записать на SQL в виде ограничения целостности, так что пусть оно существует на уровне приложения и остается невидимым для СУБД. Перевод состоит из двух операций: первая уменьшает средства на одном счете, вторая — увеличивает на другом. Первая операция нарушает согласованность данных, вторая — восстанавливает.

Если первая операция выполнится, а вторая — по причине какого-то сбоя — нет, то согласованность нарушится. А это недопустимо. Ценой неимоверных усилий такие ситуации можно обрабатывать на уровне приложения, но, к счастью, это не требуется. Задачу полностью решает СУБД, если знает, что две операции составляют неделимое целое, то есть *транзакцию*.

Но есть и второй, более тонкий момент. Транзакции, абсолютно правильные сами по себе, при одновременном выполнении могут начать работать некорректно. Это происходит из-за того, что перемешивается порядок выполнения операций разных транзакций. Если бы СУБД сначала выполняла все операции одной транзакции, а только потом — все операции другой, такой проблемы не возникало бы, но без распараллеливания работы производительность была бы невообразимо низкой.

Действительно одновременно транзакции работают в системах, где это позволяет аппаратура: многоядерный процессор, дисковый массив. Но все те же рассуждения справедливы и для сервера, который выполняет команды последовательно в режиме разделения времени. Иногда для обобщения

используют термин *конкурентное выполнение*.

Ситуации, когда корректные транзакции некорректно работают вместе, называются *аномалиями* одновременного выполнения.

Простой пример: если приложение хочет получить из базы согласованные данные, то оно как минимум не должно видеть изменения других незафиксированных транзакций. Иначе (если какая-либо транзакция будет отменена) можно увидеть состояние, в котором база данных никогда не находилась. Такая аномалия называется *грязным чтением*. Есть множество других, более сложных аномалий.

Роль СУБД состоит в том, чтобы выполнять транзакции параллельно и при этом гарантировать, что результат такого одновременного выполнения будет совпадать с результатом одного из возможных последовательных выполнений. Иными словами — *изолировать* транзакции друг от друга, устранив любые возможные аномалии.

Таким образом, транзакцией называется множество операций, которые переводят базу данных из одного корректного состояния в другое корректное состояние (*согласованность*) при условии, что транзакция выполнена полностью (*атомарность*) и без помех со стороны других транзакций (*изоляция*). Это определение объединяет требования, стоящие за первыми тремя буквами акронима ACID: Atomicity, Consistency, Isolation. Они настолько тесно связаны друг с другом, что рассматривать их по отдельности просто нет смысла. На самом деле сложно отделить и требование долговечности (Durability), ведь при крахе системы в ней остаются изменения незафиксированных транзакций, а с ними приходится что-то делать, чтобы восстановить согласованность данных.

Получается, что СУБД помогает приложению поддерживать согласованность, учитывая состав транзакции, но не имея при этом понятия о подразумеваемых правилах согласованности.

К сожалению, реализация полной изоляции — технически сложная задача, сопряженная с уменьшением производительности системы. Поэтому на

практике почти всегда применяется ослабленная изоляция, которая предотвращает некоторые, но не все аномалии. А это означает, что часть работы по обеспечению согласованности данных ложится на приложение. Именно поэтому очень важно понимать, какой уровень изоляции используется в системе, какие гарантии он дает, а какие — нет, и как в таких условиях писать корректный код.

Уровни изоляции и аномалии в стандарте SQL

Стандарт SQL описывает четыре уровня изоляции[1]. Эти уровни определяются перечислением аномалий, которые допускаются или не допускаются при одновременном выполнении транзакций. Поэтому разговор об уровнях придется начать с аномалий.

Стоит иметь в виду, что стандарт — некое теоретическое построение, которое влияет на практику, но с которым практика в то же время сильно расходится. Поэтому все примеры здесь умозрительные. Они будут использовать операции над счетами клиентов: это довольно наглядно, хотя, надо признать, не имеет ни малейшего отношения к тому, как банковские операции устроены в действительности.

Интересно, что со стандартом SQL расходится и настоящая теория баз данных, развившаяся уже после того, как стандарт был принят, а практика успела уйти вперед.

Потерянное обновление

Аномалия *потерянного обновления* (lost update) возникает, когда две транзакции читают одну и ту же строку таблицы, затем одна транзакция обновляет эту строку, после чего вторая транзакция обновляет эту же строку, не учитывая изменений, сделанных первой транзакцией.

Например, две транзакции собираются увеличить сумму на одном и том же счете на 100 ₽. Первая транзакция читает текущее значение (1000 ₽), затем вторая транзакция читает то же самое значение. Первая транзакция

увеличивает сумму (получается 1100 ₽) и записывает в базу это новое значение. Вторая транзакция поступает так же: получает те же 1100 ₽ и записывает их. В результате клиент потерял 100 ₽.

Потерянное обновление не допускается стандартом ни на одном уровне изоляции.

Грязное чтение и Read Uncommitted

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

Например, первая транзакция переводит 100 ₽ на пустой счет клиента, но не фиксирует изменение. Другая транзакция читает состояние счета (обновленное, но не зафиксированное) и позволяет клиенту снять наличные — несмотря на то, что первая транзакция прерывается и отменяет свои изменения, так что никаких денег на счете клиента нет.

Грязное чтение допускается стандартом на уровне Read Uncommitted.

Неповторяющееся чтение и Read Committed

Аномалия *неповторяющегося чтения* (non-repeatable 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.

Например, пусть правило согласованности *запрещает отрицательные суммы на счетах клиентов*. Первая транзакция собирается уменьшить сумму на счете на 100 ₽. Она проверяет текущее значение, получает 1000 ₽ и решает, что уменьшение возможно. В это время вторая транзакция уменьшает сумму на счете до нуля и фиксирует изменения. Если бы теперь первая транзакция повторно проверила сумму, она получила бы 0 ₽ (но она уже приняла решение уменьшить значение, и счет «уходит в минус»).

Неповторяющееся чтение допускается стандартом на уровнях Read

Uncommitted и Read Committed.

Фантомное чтение и Repeatable Read

Аномалия *фантомного чтения* (phantom read) возникает, когда одна транзакция два 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.

Например, пусть правило согласованности *запрещает клиенту иметь более трех счетов*. Первая транзакция собирается открыть новый счет, проверяет их текущее количество (скажем, два) и решает, что открытие возможно. В это время вторая транзакция тоже открывает клиенту новый счет и фиксирует изменения. Если бы теперь первая транзакция перепроверила количество, она получила бы три (но она уже выполняет открытие еще одного счета, и у клиента их оказывается четыре).

Фантомное чтение допускается стандартом на уровнях Read Uncommitted, Read Committed и Repeatable Read.

Отсутствие аномалий и Serializable

Стандарт определяет и уровень, на котором не допускаются никакие аномалии, — Serializable. И это совсем не то же самое, что запрет на потерянное обновление и на грязное, неповторяющееся и фантомное чтение. Дело в том, что существует значительно больше известных аномалий, чем перечислено в стандарте, и еще неизвестное число пока неизвестных.

Уровень Serializable должен предотвращать *любые* аномалии. Это означает, что на таком уровне разработчику приложения не надо думать об изоляции. Если транзакции выполняют корректные последовательности операторов, работая в одиночку, данные останутся согласованными и при одновременной работе этих транзакций.

В качестве иллюстрации приведу известную всем таблицу из стандарта, к которой для ясности добавлен последний столбец:

	потерянные изменения	грязное чтение	неповторяющееся чтение	фантомное чтение	другие аномалии
Read Uncommitted	—	да	да	да	да
Read Committed	—	—	да	да	да
Repeatable Read	—	—	—	да	да
Serializable	—	—	—	—	—

Рисунок 1 – Аномалии при выбранном режиме изоляции

Почему именно эти аномалии?

Почему из множества возможных аномалий в стандарте перечислены только несколько, и почему именно такие?

Достоверно этого, видимо, никто не знает. Но не исключено, что о других аномалиях во времена принятия первых версий стандарта просто не задумывались, поскольку теория заметно отставала от практики.

Кроме того, предполагалось, что изоляция должна быть построена на блокировках. Идея широко применявшегося *протокола двухфазного блокирования* (2PL) состоит в том, что в процессе выполнения транзакция блокирует затронутые строки, а при завершении — освобождает блокировки. Сильно упрощая: чем больше блокировок захватывает транзакция, тем лучше она изолирована от других транзакций. Но и тем сильнее страдает производительность системы, поскольку вместо совместной работы транзакции начинают выстраиваться в очередь за одними и теми же строками.

Как мне представляется, разница между стандартными уровнями изоляции в значительной степени объясняется как раз количеством необходимых для их реализации блокировок.

Если транзакция блокирует изменяемые строки для изменений, но не для чтения, получаем уровень Read Uncommitted с возможностью прочитать незафиксированные данные.

Если изменяемые строки блокируются и для чтения, и для изменений, получаем уровень Read Committed: незафиксированные данные прочитать нельзя, но при повторном обращении к строке можно получить другое

значение (неповторяющееся чтение).

Если для всех операций блокируются и читаемые, и изменяемые строки, получаем уровень Repeatable Read: повторное чтение строки будет выдавать то же значение.

Но с Serializable проблема: невозможно заблокировать строку, которой еще нет. Из-за этого остается возможность фантомного чтения: другая транзакция может добавить строку, попадающую под условия выполненного ранее запроса, и эта строка окажется в повторной выборке.

Поэтому для полной изоляции обычных блокировок не хватает — нужно блокировать не строки, а условия (предикаты). Такие *предикатные* блокировки были предложены еще в 1976 году при работе над System R, но их практическая применимость ограничена достаточно простыми условиями, для которых понятно, как объединять два разных предиката. До реализации предикатных блокировок в задуманном виде дело, насколько мне известно, не дошло ни в одной системе.

Уровни изоляции в PostgreSQL

Со временем на смену блокировочным протоколам управления транзакциями пришел *протокол изоляции на основе снимков* (Snapshot Isolation, SI). Его идея состоит в том, что каждая транзакция работает с согласованным снимком данных на определенный момент времени. В снимок попадают все актуальные изменения, зафиксированные до момента его создания.

Изоляция на основе снимков позволяет обходиться минимумом блокировок. Фактически блокируется только повторное изменение одной и той же строки. Все остальные операции могут выполняться одновременно: пишущие транзакции никогда не блокируют читающие транзакции, а читающие вообще никогда никого не блокируют.

В PostgreSQL реализован *многоверсионный* вариант протокола SI. Многоверсионность подразумевает, что в СУБД в один момент времени могут сосуществовать несколько версий одной и той же строки. Это позволяет

включать в снимок подходящую версию, а не обрывать транзакции, пытающиеся прочитать устаревшие данные.

За счет использования снимков данных изоляция в PostgreSQL отличается от той, что требует стандарт, и в целом она строже. Грязное чтение не допускается по определению. Формально в PostgreSQL можно указать уровень Read Uncommitted, но работать он будет точно так же, как Read Committed, поэтому дальше я вообще не буду говорить про этот уровень. Уровень Repeatable Read не допускает не только неповторяющегося, но и фантомного чтения (хотя и не обеспечивает полную изоляцию). Правда, на уровне Read Committed можно *в ряде случаев* потерять изменения.

	потерянные изменения	грязное чтение	неповторяющееся чтение	фантомное чтение	другие аномалии
Read Committed	да	—	да	да	да
Repeatable Read	—	—	—	—	да
Serializable	—	—	—	—	—

Рисунок 2 – Аномалии при выбранном режиме изоляции в PostgreSQL

Перед тем как исследовать внутреннее устройство изоляции, давайте подробно рассмотрим каждый из трех уровней с точки зрения пользователя.

Для этого создадим таблицу счетов. У Алисы и Боба по 1000 ₽, но у Боба открыто два счета:

```
=>CREATE TABLE accounts(
id integer PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY,
client text, amount numeric
);
=> INSERT INTO accounts VALUES
(1, 'alice', 1000.00),
(2, 'bob', 100.00),
(3, 'bob', 900.00);
```

Read Committed

Отсутствие грязного чтения. Легко убедиться в том, что грязные данные прочитать невозможно. Начнем транзакцию. По умолчанию она использует уровень изоляции Read Committed[1]:

```
=> BEGIN;
=> SHOW transaction_isolation; transaction_isolation
-----------------------
read committed (1 row)
```

Говоря точнее, умолчательный уровень задается параметром, который при необходимости можно изменить:

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

Итак, в открытой транзакции снимаем средства со счета, но не фиксируем изменения. Свои собственные изменения транзакция всегда видит:

```
=> UPDATE accounts SET amount = amount 200 WHERE id
= 1;
=> SELECT * FROM accounts WHERE client = 'alice'; id
| client | amount
----+--------+--------
1 | alice | 800.00 (1 row)
```

Во втором сеансе начинаем еще одну транзакцию с тем же уровнем Read Committed:

```
=> BEGIN;
```

```
    => SELECT * FROM accounts WHERE client = 'alice'; id
| client | amount
    ----+--------+---------
    1 | alice | 1000.00
    (1 row)
```

Как и ожидалось, другая транзакция не видит незафиксированные изменения — грязное чтение не допускается.

Неповторяющееся чтение. Пусть теперь первая транзакция зафиксирует изменения, а вторая повторно выполнит тот же самый запрос.

```
    => COMMIT;
```

```
    => SELECT * FROM accounts WHERE client = 'alice'; id
| client | amount
    ----+--------+--------
    1 | alice | 800.00 (1 row)
    => COMMIT;
```

Запрос получает уже новые данные— это и есть аномалия *неповторяющегося чтения*, которая допускается на уровне Read Committed.

Практический вывод: в транзакции нельзя принимать решения на основании данных, прочитанных предыдущим оператором, ведь за время между выполнением операторов все может измениться. Вот пример, вариации которого встречаются в прикладном коде так часто, что он является классическим антипаттерном:

```
    IF (SELECT amount FROM accounts WHERE id = 1) >=
1000 THEN UPDATE accounts SET amount = amount 1000 WHERE
id = 1;
    END IF;
```

За время, которое проходит между проверкой и обновлением, другие транзакции могут как угодно изменить состояние счета, так что такая «проверка» ни от чего не спасает. Удобно представлять себе, что между операторами одной транзакции могут «вклиниться» произвольные операторы других транзакций, например вот так:

```
IF (SELECT amount FROM accounts WHERE id = 1) >= 1000 THEN

    UPDATE accounts SET amount = amount 200 WHERE id = 1;
    COMMIT;

    UPDATE accounts SET amount = amount 1000 WHERE id = 1;
    END IF;
```

Если, переставив операторы, можно все испортить, значит, код написан некорректно. Не стоит обманывать себя, что с таким стечением обстоятельств мы не столкнемся: если неприятность может случиться, она произойдет обязательно. А вот воспроизводить и, следовательно, исправлять такие ошибки очень сложно.

Как написать код корректно? Есть несколько возможностей:

- Заменить процедурный код декларативным.

  Например, в данном случае проверка легко превращается в ограничение целостности:

```
ALTER TABLE accounts ADD CHECK amount >= 0;
```

  Теперь никакие проверки в коде не нужны: достаточно просто выполнить действие и при необходимости обработать исключение,

которое возникнет в случае попытки нарушения целостности.

- Использовать один оператор SQL.

Проблемы с согласованностью возникают из-за того, что в промежутке между операторами может завершиться другая транзакция, и видимые данные изменятся. Если оператор один, то и промежутков никаких нет.

В PostgreSQL достаточно средств, чтобы одним SQL-оператором решать сложные задачи. Отмечу общие табличные выражения (CTE), в которых в том числе можно использовать операторы INSERT, UPDATE, DELETE, а также оператор INSERT ON CONFLICT, который атомарно реализует логику «вставить, а если строка уже есть, то обновить».

- Задействовать пользовательские блокировки.

Последнее средство — вручную установить исключительную блокировку на все нужные строки (SELECT FOR UPDATE) или вообще на всю таблицу (LOCK TABLE). Это всегда работает, но сводит на нет преимущества многоверсионности: вместо одновременного выполнения часть операций будет выполняться последовательно.

Несогласованное чтение. Однако не все так просто. Реализация PostgreSQL такова, что допускает другие, менее известные аномалии, которые не регламентируются стандартом.

Допустим, первая транзакция начала перевод средств с одного счета Боба на другой:

```
=> BEGIN;
=> UPDATE accounts SET amount = amount 100 WHERE id = 2;
```

В это время другая транзакция подсчитывает баланс Боба, причем подсчет выполняется в цикле по всем счетам Боба. Фактически транзакция начинает с первого счета (и, разумеется, видит прежнее состояние):

```
=> BEGIN;
=> SELECT amount FROM accounts WHERE id = 2; amount
-------- 100.00
(1 row)
```

В этот момент первая транзакция успешно завершается:

```
=> UPDATE accounts SET amount = amount + 100 WHERE
id = 3;
=> COMMIT;
```

А другая читает состояние второго счета (и видит уже новое значение):

```
=> SELECT amount FROM accounts WHERE id = 3; amount
--------- 1000.00
(1 row)
=> COMMIT;
```

В итоге вторая транзакция получила в сумме 1100 ₽, то есть прочитала некорректные данные. Такая аномалия называется *несогласованным чтением* (read skew).

Как избежать этой аномалии, оставаясь на уровне Read Committed? Конечно, использовать один оператор. Например, так:

```
SELECT sum(amount) FROM accounts WHERE client =
'bob';
```

До сих пор я утверждал, что видимость данных может поменяться только между операторами, но так ли это очевидно? А если запрос выполняется долго, может ли он увидеть часть данных в одном состоянии, а часть — уже в другом?

Проверим. Удобный способ для этого — вставить в оператор

искусственную задержку, вызвав функцию pg_sleep. Первая строка будет прочитана сразу, а вторая — через две секунды:

```
=> SELECT amount, pg_sleep(2) -2 секунды
FROM accounts WHERE client = 'bob';
```

Пока эта конструкция выполняется, в другой транзакции переводим средства обратно:

```
=> BEGIN;
=> UPDATE accounts SET amount = amount + 100 WHERE
id = 2;
=> UPDATE accounts SET amount = amount 100 WHERE id
= 3;
=> COMMIT;
```

Результат показывает, что оператор видит данные в таком состоянии, в каком они находились на момент начала его выполнения, что, безусловно, правильно:

```
amount | pg_sleep
---------+----------
0.00 | 1000.00 |
(2 rows)
```

Но и тут не все так просто. Если в запросе вызывается *изменчивая* функция (с категорией изменчивости VOLATILE) и в этой функции выполняется другой запрос, то этот вложенный запрос будет видеть данные, не согласованные с данными основного запроса.

Проверим состояние счетов Боба, используя функцию:

```
=> CREATE FUNCTION get_amount(id integer) RETURNS
```

```
numeric
    AS $$
    SELECT amount FROM accounts a WHERE a.id =
get_amount.id;
    $$ VOLATILE LANGUAGE sql;
    => SELECT get_amount(id), pg_sleep(2)
    FROM accounts WHERE client = 'bob';
```

И снова переведем деньги между счетами, пока запрос с задержкой выполняется:

```
    => BEGIN;
    => UPDATE accounts SET amount = amount + 100 WHERE
id = 2;
    => UPDATE accounts SET amount = amount 100 WHERE id
= 3;
    => COMMIT;
```

В этом случае получим несогласованные данные — 100 ₽ пропали:

```
    get_amount | pg_sleep
    ------------+----------
    100.00 |
    800.00 | (2 rows)
```

Подчеркну, что такой эффект возможен только на уровне изоляции Read Committed и только с категорией изменчивости VOLATILE. Беда в том, что по умолчанию используется именно этот уровень изоляции и именно эта категория изменчивости, так что остается признать — грабли лежат очень удачно.

Несогласованное чтение вместо потерянного обновления.

Аномалию несогласованного чтения в рамках одного оператора можно — несколько неожиданным образом — получить и при обновлении.

Посмотрим, что происходит при попытке изменения одной и той же строки двумя транзакциями. Сейчас у Боба 1000 ₽ на двух счетах:

```
=> SELECT * FROM accounts WHERE client = 'bob'; id | client | amount
----+--------+--------
2 | bob  | 200.00
3 | bob  | 800.00
(2 rows)
```

Начинаем транзакцию, которая уменьшает баланс Боба:

```
=> BEGIN;
=> UPDATE accounts SET amount = amount 100 WHERE id = 3;
```

В это же время другая транзакция начисляет проценты на все счета с общим балансом, равным или превышающим 1000 ₽:

```
=> UPDATE accounts SET amount = amount * 1.01
WHERE client IN ( SELECT client FROM accounts GROUP BY client
HAVING sum(amount) >= 1000
);
```

Выполнение оператора UPDATE состоит из двух частей. Сначала фактически выполняется оператор SELECT, который отбирает для обновления строки, соответствующие условию. Поскольку изменение первой транзакции не зафиксировано, вторая транзакция не может его видеть, и оно никак не влияет на выбор строк для начисления процентов. Таким образом, счета Боба

попадают под условие, и после выполнения обновления его баланс должен увеличиться на 10 ₽.

На втором этапе выполнения выбранные строки обновляются одна за другой. Вторая транзакция вынуждена подождать, поскольку строка id = 3 в настоящий момент изменяется первой транзакцией и поэтому заблокирована.

Между тем первая транзакция фиксирует изменения:

```
=> COMMIT;
=> SELECT * FROM accounts WHERE client = 'bob'; id | client | amount
----+--------+----------
2 | bob  | 202.0000
3 | bob  | 707.0000
(2 rows)
```

Да, с одной стороны, команда UPDATE не должна видеть изменений второй транзакции. Но с другой — она не должна потерять изменения, зафиксированные во второй транзакции.

После снятия блокировки оператор UPDATE *перечитывает* строку, которую пытается обновить (но только ее одну!). В результате получается, что Бобу начислено 9 ₽, исходя из суммы 900 ₽. Но если бы у Боба было 900 ₽, его счета вообще не должны были попасть в выборку.

Таким образом, транзакция прочитала некорректные данные: часть строк — на один момент времени, часть — на другой. Взамен потерянного обновления мы снова получаем аномалию несогласованного чтения.

Потерянное обновление. Впрочем, хитрость с перечитыванием заблокированной строки не спасает от потери изменений, если обновление происходит не в одном операторе SQL.

Вот пример, который уже был. Приложение читает и запоминает (вне базы данных) текущий баланс счета Алисы:

```
=> BEGIN;
=> SELECT amount FROM accounts WHERE id = 1; amount
-------- 800.00
(1 row)
```

В это время другая транзакция действует так же:

```
=> BEGIN;
=> SELECT amount FROM accounts WHERE id = 1; amount
-------- 800.00
(1 row)
```

Первая транзакция увеличивает запомненное ранее значение на 100 ₽ и записывает в базу:

```
=> UPDATE accounts SET amount = 800.00 + 100 WHERE
id = 1
RETURNING amount;
amount
-------- 900.00
(1 row)
UPDATE 1
=> COMMIT;
```

И вторая транзакция тоже:

```
=> UPDATE accounts SET amount = 800.00 + 100 WHERE
id = 1
RETURNING amount;
amount
-------- 900.00
(1 row)
```

```
UPDATE 1
=> COMMIT;
```

К сожалению, Алиса недосчиталась 100 ₽. СУБД ничего не знает о том, что запомненное значение 800 ₽ как-то связано с accounts.amount, и допускает аномалию потерянного изменения. На уровне изоляции Read Committed такой код некорректен.

Repeatable Read

Отсутствие неповторяющегося и фантомного чтения. Само название уровня изоляции Repeatable Read[1] говорит о повторяемости чтения. Проверим это, а заодно убедимся и в отсутствии фантомов. Для этого в первой транзакции вернем счета Боба в прежнее состояние и создадим новый счет для Чарли:

```
=> BEGIN;
=> UPDATE accounts SET amount = 200.00 WHERE id = 2;
=> UPDATE accounts SET amount = 800.00 WHERE id = 3;
=> INSERT INTO accounts VALUES
(4, 'charlie', 100.00);
=> SELECT * FROM accounts ORDER BY id; id | client |
amount
----+---------+--------
1 | alice   | 900.00 2 | bob | 200.00
3 | bob | 800.00
4 | charlie | 100.00 (4 rows)
```

Во втором сеансе начнем транзакцию с уровнем Repeatable Read, указав его в команде BEGIN (уровень первой транзакции не важен):

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
```

```
    => SELECT * FROM accounts ORDER BY id; id | client |
amount
    ----+--------+----------
    1 | alice | 900.00 2 | bob    | 202.0000 3 | bob |
707.0000
    (3 rows)
```

Теперь первая транзакция фиксирует изменения, а вторая повторно выполняет тот же самый запрос:

```
    => COMMIT;

    => SELECT * FROM accounts ORDER BY id; id | client |
amount
    ----+--------+----------
    1 | alice | 900.00 2 | bob    | 202.0000 3 | bob |
707.0000
    (3 rows)
    => COMMIT;
```

Вторая транзакция продолжает видеть ровно те же данные, что и в начале: не видно ни изменений в существующих строках, ни новых строк. На таком уровне можно не беспокоиться о том, что между двумя операторами что-то поменяется.

Ошибка сериализации вместо потерянных изменений. Как мы уже видели, на уровне изоляции Read Committed при обновлении одной и той же строки двумя транзакциями может возникнуть аномалия несогласованного чтения. Это происходит из-за того, что ожидающая транзакция перечитывает заблокированную строку и видит ее на один момент времени, а остальные строки выборки — на другой.

На уровне Repeatable Read такая аномалия не допускается, но если она

всетаки возникает, сделать уже ничего нельзя — поэтому транзакция обрывается с ошибкой сериализации. Проверим, повторив тот же сценарий с процентами:

```
=> SELECT * FROM accounts WHERE client = 'bob'; id | client | amount
----+--------+--------
2 | bob  | 200.00
3 | bob  | 800.00
(2 rows)
=> BEGIN;
=> UPDATE accounts SET amount = amount 100.00 WHERE id = 3;

=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> UPDATE accounts SET amount = amount * 1.01
WHERE client IN ( SELECT client FROM accounts GROUP BY client
HAVING sum(amount) >= 1000
);

=> COMMIT;

ERROR: could not serialize access due to concurrent update
=> ROLLBACK;
```

Данные остались согласованными:

```
=> SELECT * FROM accounts WHERE client = 'bob'; id | client | amount
----+--------+--------
2 | bob  | 200.00
3 | bob  | 700.00
```

```
(2 rows)
```

Такая же ошибка будет и в случае любого другого конкурентного изменения строки, даже если оно не затрагивает интересующие нас столбцы.

Можно проверить, что и сценарий с приложениями, обновляющими баланс на основе запомненного значения, приводит к той же ошибке:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> SELECT amount FROM accounts WHERE id = 1; amount
-------- 900.00
(1 row)

=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> SELECT amount FROM accounts WHERE id = 1; amount
-------- 900.00
(1 row)


=> UPDATE accounts SET amount = 900.00 + 100.00
WHERE id = 1
RETURNING amount;
amount
--------- 1000.00
(1 row)
UPDATE 1
=> COMMIT;

=> UPDATE accounts SET amount = 900.00 + 100.00
WHERE id = 1
RETURNING amount;
ERROR: could not serialize access due to concurrent
update
```

```
=> ROLLBACK;
```

Практический вывод: если приложение использует уровень изоляции Repeatable Read для пишущих транзакций, оно должно быть готово повторять транзакции, завершившиеся ошибкой сериализации. Для только читающих транзакций такой исход невозможен.

Несогласованная запись. Итак, в PostgreSQL на уровне изоляции Repeatable Read предотвращаются все аномалии, описанные в стандарте. Но не все вообще — науке до сих пор неизвестно, сколько их существует. Однако доказан важный факт: изоляция на основе снимков оставляет возможными *ровно две* аномалии, сколько бы их ни было всего.

Первая из этих аномалий — *несогласованная запись* (write skew).

Пусть действует такое правило согласованности: *допускаются отрицательные суммы на отдельных счетах клиента, если общая сумма на всех счетах этого клиента остается неотрицательной*.

Первая транзакция получает сумму на счетах Боба:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> SELECT sum(amount) FROM accounts WHERE client =
'bob'; sum
-------- 900.00
(1 row)
```

Вторая транзакция получает ту же сумму:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> SELECT sum(amount) FROM accounts WHERE client =
'bob'; sum
-------- 900.00
(1 row)
```

Первая транзакция справедливо полагает, что сумму одного из счетов можно уменьшить на 600 ₽:

```
=> UPDATE accounts SET amount = amount 600.00 WHERE id = 2;
```

И вторая транзакция приходит к такому же выводу. Но уменьшает другой счет:

```
=> UPDATE accounts SET amount = amount 600.00 WHERE id = 3;
=> COMMIT;
```

```
=> COMMIT;
=> SELECT * FROM accounts WHERE client = 'bob'; id | client | amount
----+--------+---------
2 | bob  | -400.00
3 | bob  | 100.00
(2 rows)
```

У нас получилось увести баланс Боба в минус, хотя поодиночке каждая из транзакций отработала бы корректно.

Аномалия только читающей транзакции. Аномалия *только читающей транзакции* — вторая и последняя из аномалий, возможных на уровне Repeatable Read. Чтобы продемонстрировать ее, потребуются три транзакции, две из которых будут изменять данные, а третья — только читать.

Но сначала восстановим состояние счетов Боба:

```
=> UPDATE accounts SET amount = 900.00 WHERE id = 2;
```

```
=> SELECT * FROM accounts WHERE client = 'bob';
id | client | amount
----+--------+--------
3 | bob  | 100.00
2 | bob  | 900.00
(2 rows)
```

Первая транзакция начисляет Бобу проценты на сумму средств на всех счетах. Проценты зачисляются на один из его счетов:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ; -1
=> UPDATE accounts SET amount = amount + (
SELECT sum(amount) FROM accounts WHERE client =
'bob'
) * 0.01
WHERE id = 2;
```

Затем другая транзакция снимает деньги с другого счета Боба и фиксирует свои изменения:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ; -2
=> UPDATE accounts SET amount = amount 100.00 WHERE
id = 3;
=> COMMIT;
```

Если в этот момент первая транзакция будет зафиксирована, никакой аномалии не возникнет: мы могли бы считать, что сначала выполнена первая транзакция, а затем вторая (но не наоборот, потому что первая транзакция увидела состояние счета id = 3 до того, как этот счет был изменен второй транзакцией).

Но представим, что в этот момент начинается третья (только читающая) транзакция, которая получает состояние какого-нибудь счета, не затронутого

первыми двумя транзакциями:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ; -3
=> SELECT * FROM accounts WHERE client = 'alice'; id
| client | amount
----+--------+---------
1 | alice | 1000.00
(1 row)
```

И только после этого первая транзакция завершается:

```
=> COMMIT;
```

Какое состояние теперь должна увидеть третья транзакция? Начавшись, она могла видеть изменения второй транзакции (которая уже была зафиксирована), но не первой (которая еще не была зафиксирована). С другой стороны, мы уже установили выше, что вторую транзакцию следует считать начавшейся после первой. Какое состояние ни увидит третья транзакция, оно будет несогласованным — это и есть аномалия только читающей транзакции:

```
=> SELECT * FROM accounts WHERE client = 'bob'; id |
client | amount
----+--------+--------
2 | bob  | 900.00
3 | bob  |   0.00 (2 rows)
=> COMMIT;
```

Точки сохранения

Полезность свойства атомарности состоит в том, что в случае неудачного выполнения некоторой транзакции частичные изменения, выполненные этой транзакцией, удаляются из базы данных и не мешают работе других

транзакций. Следовательно, каждое приложение вправе ожидать, что оно начинает работу с корректным (согласованным) состоянием базы данных. Однако ресурсы, затраченные на выполнение оборванной транзакции, оказываются безвозвратно потерянными. Это вполне допустимо для коротких транзакций, на выполнение каждой из которых затрачивается небольшое количество ресурсов, и если аварийные завершения случаются редко. В противоположность этому потеря результатов выполнения работы большого объема нежелательна.

Для транзакций, выполняющих большие объемы работы, может быть целесообразно использовать оператор SAVEPOINT, который запоминает состояние всех данных, измененных транзакцией к моменту его выполнения. В случае если при продолжении выполнения транзакции обнаруживается ошибка, можно восстановить состояние базы данных, выполняя операцию ROLLBACK с указанием имени того состояния, которое было предварительно записано оператором SAVEPOINT.

В следующем простом примере иллюстрируется работа операторов создания точек сохранения. В рамках транзакции выполняется изменение строки и создается точка сохранения; затем эта строка удаляется. После отката к точке сохранения с помощью оператора ROLLBACK вставленная строка снова появляется в базе данных. Наконец, вся транзакция обрывается — база данных восстанавливается в состояние, эквивалентное тому, которое было до начала транзакции.

```
demo=# BEGIN TRANSACTION; BEGIN
demo=# SELECT * FROM aircrafts
WHERE aircraft_code = '320';
aircraft_code |   model    | range
--------------+------------------+-------
320  | Аэробус A320-200 | 5700 (1 row)
demo=# UPDATE aircrafts SET range = 6200
WHERE aircraft_code = '320'; UPDATE 1
```

```
demo=# SELECT * FROM aircrafts
WHERE aircraft_code = '320';
aircraft_code |  model    | range
--------------+-----------------+-------
320  | Аэробус A320-200 | 6200 (1 row)
demo=# SAVEPOINT svp; SAVEPOINT
demo=#  DELETE  FROM  aircrafts  WHERE  aircraft_code  =
'320';
DELETE 1
demo=# ROLLBACK TO svp; ROLLBACK
demo=# SELECT * FROM aircrafts
WHERE aircraft_code = '320';
aircraft_code |  model    | range
--------------+-----------------+-------
320  | Аэробус A320-200 | 6200 (1 row)
demo=# ROLLBACK;
ROLLBACK
demo=# SELECT * FROM aircrafts
WHERE aircraft_code = '320';
aircraft_code |  model    | range
--------------+-----------------+-------
320  | Аэробус A320-200 | 5700 (1 row)
```

Механизм точек сохранения дает возможность организовать достаточно сложное поведение в рамках одной транзакции, но для других транзакций ее поведение остается атомарным.

Задание на практическую работу 2:

1. Начните транзакцию (командой BEGIN) и создайте новый заказ в таблице sales_order с сегодняшней датой. Добавьте два предмета в таблицу item, связанных с созданным заказом.

Представьте, что пользователь не подтвердил заказ и все введенные данные необходимо отменить. Выполните отмену транзакции и проверьте, что никакой добавленной вами информации действительно не осталось.

2. Теперь представьте сценарий, в котором нужно отменить не все данные, а только последний из добавленных предметов. Для этого повторите все действия из предыдущего упражнения, но перед добавлением каждого предмета создавайте точку сохранения (с одним и тем же именем). После ввода второго предмета выполните откат к точке сохранения. Проверьте, что заказ и первый предмет остались.

3. В рамках той же транзакции добавьте еще один предмет и зафиксируйте транзакцию. Обратите внимание на то, что после этой операции отменить внесенные транзакцией изменения будет уже невозможно.

Уровень изоляции Read Committed

4. Перед началом выполнения задания проверьте, что в таблице sales_order нет заказов на сумму total 1 000 рублей.

4.1. В первом сеансе начните транзакцию (командой BEGIN). Выполните обновление таблицы sales_order: увеличьте total в два раза в тех строках, где сумма равна 1 000 рублей.

4.2. Во втором сеансе (откройте новое окно psql) вставьте в таблицу sales_order новый заказ на 1 000 рублей и зафиксируйте транзакцию.

4.3. В первом сеансе повторите обновление таблицы sales_order и зафиксируйте транзакцию.

Осталась ли сумма добавленного заказа равной 1 000 рублей? Почему это не так?

Уровень изоляции Repeatable Read

5. Повторите предыдущее упражнение, но начните транзакцию в первом сеансе с уровнем изоляции транзакций Repeatable Read. Объясните различие полученных результатов.

6. Выполните указанные действия в двух сеансах:

6.1. В первом сеансе начните новую транзакцию с уровнем

изоляции Repeatable Read. Вычислите количество заказов с суммой 20 000 рублей.

6.2. Во втором сеансе начните новую транзакцию с уровнем изоляции Repeatable Read. Вычислите количество заказов с суммой 30 000 рублей.

6.3. В первом сеансе добавьте новый заказ на 30 000 рублей и снова вычислите количество заказов с суммой 20 000 рублей.

6.4. Во втором сеансе добавьте новый заказ на 20 000 рублей и снова вычислите количество заказов с суммой 30 000 рублей.

6.5. Зафиксируйте транзакции в обоих сеансах.

Соответствует ли результат ожиданиями? Можно ли сериализовать эти транзакции (иными словами, можно ли представить такой порядок последовательного выполнения этих транзакций, при котором результат совпадет с тем, что получился при параллельном выполнении)?