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

## Практическая работа №3

## Транзакции. Уровни изоляции. Уровень изоляции Serializable в PostgreSQL.

### Теория для понимания практики, желательна к выполнению:

### Уровень изоляции Serializable

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

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

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

```
CREATE TABLE
```

Добавим в таблицу две строки.

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

```
INSERT 0 2
```

Итак, содержимое таблицы имеет вид:

```
SELECT * FROM modes;

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

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

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN
```

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

```
UPDATE modes
     SET mode = 'HIGH'
     WHERE mode = 'LOW'
     RETURNING *;
num | mode
-----+------
1 | HIGH
(1 строка)
UPDATE 1
```

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

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;

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

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

UPDATE 1
```

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

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

Посмотрим, что получилось в первой транзакции:

```
SELECT * FROM modes;

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

А во второй транзакции:

```
SELECT * FROM modes;

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

Заканчиваем эксперимент. Сначала завершим транзакцию на первом терминале:

```
COMMIT;
COMMIT
```

А потом на втором терминале:

```
COMMIT;

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

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

```
SELECT * FROM modes;

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

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

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

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

Покажем это, выполнив транзакции последовательно.

Предварительно необходимо пересоздать таблицу modes или с помощью команды UPDATE вернуть ее измененным строкам исходное состояние. Теперь обе транзакции можно выполнять на одном терминале. *Команда END равнозначна команде COMMIT.*

Первый вариант их упорядочения такой:

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN

UPDATE modes
     SET mode = 'HIGH'
     WHERE mode = 'LOW'
     RETURNING *;
num | mode
-----+------
1 | HIGH
(1 строка)
UPDATE 1
END;
COMMIT
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN

UPDATE modes
     SET mode = 'LOW'
     WHERE mode = 'HIGH'
     RETURNING *;
num | mode
-----+------
2 | LOW
1 | LOW
(2 строки)
```

```
UPDATE 2
END;
COMMIT
```

Проверим, что получилось:

```
SELECT * FROM modes;

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

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

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN

UPDATE modes
     SET mode = 'LOW'
     WHERE mode = 'HIGH'
     RETURNING *;
num | mode
-----+------
2| LOW(1 строка)

UPDATE 1
END;
COMMIT
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN

UPDATE modes
```

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

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

UPDATE 2
END;
COMMIT
SELECT * FROM modes;
```

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

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

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

На уровне Serializable предотвращаются все возможные аномалии. Фактически Serializable реализован как надстройка над изоляцией на основе снимков данных. Те аномалии, которые не возникают при Repeatable Read (такие как грязное, неповторяемое, фантомное чтение), не возникают и на

уровне Serializable. А те две аномалии, которые возникают (несогласованная запись и аномалия только читающей транзакции), специальным образом обнаруживаются, и при необходимости транзакция обрывается: возникает уже знакомая нам ошибка сериализации.

Отсутствие аномалий. Можно убедиться, что сценарий с аномалией несогласованной записи приведет к ошибке сериализации:

```
BEGIN ISOLATION LEVEL SERIALIZABLE;
SELECT sum(amount) FROM accounts WHERE client = 'bob';
sum
---------- 910.0000
(1 row)

BEGIN ISOLATION LEVEL SERIALIZABLE;
SELECT sum(amount) FROM accounts WHERE client = 'bob';
sum
---------- 910.0000
(1 row)

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

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

COMMIT;
ERROR:  could not serialize access due to read/write dependencies
among transactions
DETAIL:  Reason code: Canceled on identification as a pivot,
during commit attempt.
HINT:  The transaction might succeed if retried.
```

К такой же ошибке приведет и сценарий аномалии только читающей транзакции.

Откладывание читающей транзакции. Чтобы только читающая транзакция не приводила к аномалии и не могла пострадать от нее, PostgreSQL предлагает интересный механизм: такая транзакция может быть отложена до тех пор, пока ее выполнение не станет безопасным. Это единственный случай, когда оператор SELECT может быть заблокирован обновлениями строк.

Посмотрим на примере сценария аномалии только читающей транзакции:

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

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

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

Третью транзакцию явно объявляем только читающий (READ ONLY) и откладываемой (DEFERRABLE):

```
BEGIN ISOLATION LEVEL SERIALIZABLE READ ONLY DEFERRABLE; -- 3
SELECT * FROM accounts WHERE client = 'alice';
```

При попытке выполнить запрос транзакция блокируется, потому что иначе она приведет к аномалии.

И только после того, как первая транзакция будет зафиксирована, третья продолжит выполнение:

```
COMMIT;

id | client | amount
----+--------+---------
1 | alice  | 1000.00
(1 row)
SELECT * FROM accounts WHERE client = 'bob';
id | client |  amount
----+--------+----------
2 | bob    | 910.0000
3 | bob    |    0.00
(2 rows)
COMMIT;
```

Таким образом, приложение, использующее уровень изоляции Serializable, должно повторять транзакции, завершившиеся ошибкой сериализации. (Точно так же следует поступать и на уровне Repeatable Read, если не ограничиваться только читающими транзакциями.)

Уровень Serializable дает простоту программирования, но цена за нее — накладные расходы на обнаружение возможных аномалий и обрыв некоторой доли транзакций. Снизить накладные расходы можно, явно обозначая только читающие транзакции как `READ ONLY`. Но главный вопрос, конечно, в том, насколько велика доля оборванных транзакций, ведь их придется выполнять повторно. Если бы обрывались только те транзакции, которые действительно несовместимо пересекаются по данным, все было бы неплохо. Но такая реализация неизбежно оказалась бы слишком ресурсоемкой, поскольку пришлось бы отслеживать операции с каждой строкой.

В действительности реализация такова, что допускает

ложноотрицательные срабатывания: будут обрываться и некоторые совершенно нормальные транзакции, которым «просто не повезло». Везение зависит от многих причин, например от наличия подходящих индексов или доступного объема оперативной памяти, и поведение сложно предсказать заранее.

Если используется уровень изоляции Serializable, он должен применяться для всех транзакций приложения. При смешении транзакций разного уровня изоляции уровень Serializable будет (без всяких предупреждений) вести себя как Repeatable Read. Поэтому при использовании Serializable имеет смысл изменить значение параметра *default_transaction_isolation*, хотя, конечно, это не помешает указать неправильный уровень явно.

Есть и другие ограничения реализации, например запросы на уровне Serializable не будут работать на репликах. И хотя работа над улучшением функциональности не прекращается, имеющиеся ограничения и накладные расходы снижают привлекательность такого уровня изоляции.

**Какой уровень изоляции использовать?**

Уровень изоляции Read Committed используется в PostgreSQL по умолчанию, и, по всей видимости, именно этот уровень применяется в абсолютном большинстве приложений. Он удобен тем, что на нем обрыв транзакции возможен только в случае сбоя, но для предотвращения несогласованности обрыв не применяется. Иными словами, ошибка сериализации возникнуть не может, и о повторении транзакций заботиться не надо.

Обратная сторона медали — большое число возможных аномалий, подробно рассмотренных выше. Разработчик вынужден постоянно иметь их в виду и писать код так, чтобы не допускать их появления. Если не получается сформулировать нужные действия в одном SQL-операторе, приходится прибегать к явной установке блокировок. Самое неприятное то, что код сложно тестировать на наличие ошибок, связанных с получением

несогласованных данных, а сами ошибки могут возникать непредсказуемым и невоспроизводимым образом и поэтому сложны в исправлении.

Уровень изоляции Repeatable Read снимает часть проблем несогласованности, но, увы, не все. Поэтому приходится не только помнить об оставшихся аномалиях, но и изменять приложение так, чтобы оно корректно обрабатывало ошибки сериализации. Это, конечно, неудобно. Но для только читающих транзакций этот уровень прекрасно дополняет Read Committed и полезен, например, для построения отчетов, использующих несколько SQL запросов.

Наконец, уровень Serializable позволяет вообще не заботиться о несогласованности, что значительно упрощает написание кода. Единственное, что требуется от приложения, — уметь повторять любую транзакцию при получении ошибки сериализации. Но доля прерываемых транзакций и дополнительные накладные расходы могут существенно снизить пропускную способность. Также следует учитывать, что уровень Serializable не применим на репликах и что его нельзя смешивать с другими уровнями изоляции.

**Задание на практическую работу:**

1. В первом сеансе начните новую транзакцию с уровнем изоляции Serializable. Вычислите количество заказов с суммой 20 000 рублей.

2. Во втором сеансе начните новую транзакцию с уровнем изоляции Serializable. Вычислите количество заказов с суммой 30 000 рублей.

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

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

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)?

Если вы правильно ответили на его последний вопрос, вы поймете, почему теперь эти действия приводят к ошибке. Если же результат этого упражнения стал для вас неожиданностью, четко сформулируйте различие уровней Repeatable Read и Serializable.