ГУАП

КАФЕДРА № 53

ОТЧЕТ   
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| доцент, канд. техн. наук |  |  |  | А. В. Бржезовский |
| должность, уч. степень, звание |  | подпись, дата |  | инициалы, фамилия |

|  |
| --- |
| ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ |
| Транзакции и блокировки |
| по курсу: методы и средства проектирования информационных систем и технологий |
|  |
|  |

РАБОТУ ВЫПОЛНИЛ

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| СТУДЕНТ ГР. № | 3641 |  |  |  | М.Б.Фомин |
|  |  |  | подпись, дата |  | инициалы, фамилия |

Санкт-Петербург 2020

*Задание:*

Смоделировать в БД

грязное чтение, неповторяемое чтение, фантомы, изменяя уровень изоляции транзакций продемонстрировать их исключение, сформировать отчеты о блокировках, пояснить их содержание.

Смоделировать в БД тупик (взаимную блокировку), получить с помощью приложения

*Выполнение работы:*

1. **Уровни изоляции и аномалии**
   1. **Уровни изоляции и аномалии в стандарте SQL**

Стандарт SQL с давних пор описывает четыре уровня изоляции. Эти уровни определяются перечислением аномалий, которые допускаются или не допускаются при одновременном выполнении транзакций на этом уровне.



Рисунок 1:Уровни изоляции и аномалии в стандарте SQL.

Предполагалось, что изоляция должна быть построена на блокировках. Идея широко применявшегося протокола двухфазного блокирования (2PL) состоит в том, что в процессе выполнения транзакция блокирует строки, с которыми работает, 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.

* 1. **Уровни изоляции в PostgreSQL**

Со временем на смену блокировочным протоколам управления транзакциями пришел протокол изоляции на основе снимков (Snapshot Isolation). Его идея состоит в том, что каждая транзакция работает с согласованным снимком данных на определенный момент времени, в который попадают только те изменения, которые были зафиксированы до момента создания снимка.

Такая изоляция автоматически не допускает грязное чтение. Формально в PostgreSQL можно указать уровень Read Uncommitted, но работать она будет точно так же, как Read Committed.

В PostgreSQL реализован многоверсионный вариант такого протокола. Идея многоверсионности состоит в том, что в СУБД могут сосуществовать несколько версий одной и той же строки. Это позволяет строить снимок данных, используя имеющиеся версии, и обходиться минимумом блокировок. Фактически блокируется только повторное изменение одной и той же строки. Все остальные операции выполняются одновременно: пишущие транзакции никогда не блокируют читающих транзакций, а читающие никогда не блокируют никого.

За счет использования снимков данных изоляция в PostgreSQL получается строже, чем того требует стандарт: уровень Repeatable Read не допускает не только неповторяющегося, но и фантомного чтения (хотя и не обеспечивает полную изоляцию). И достигается это без потери эффективности.



Рисунок 2:Уровни изоляции и аномалии в Postgresql.

1. **Моделирование аномалий в Postgresql**
   1. **dirty read — «грязное» чтение**

Легко убедиться в том, что грязные данные прочитать невозможно. Начнём транзакцию изменяющую количество агентов, у которых процент с продаж меньше 10.

Уровень изоляции транзакций READ UNCOMMITTED.

**Транзакция 1(pid = 8428)**

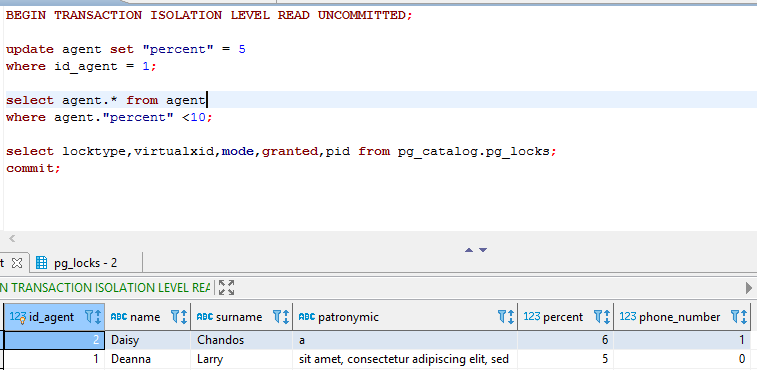


Рисунок 3: незафиксированные изменения в транзакции 1

**Транзакция 2(pid = 7468)**

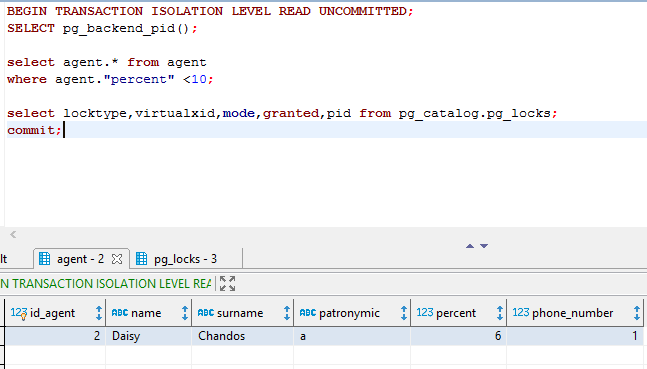


Рисунок 4:результат работы транзакции 2

**Блокировки транзакций**

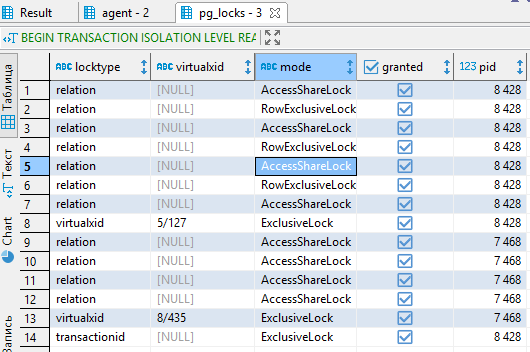


Рисунок 5: блокировки транзакций

* 1. **non-repeatable read — неповторяемое чтение**

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

Смоделируем неповторяемое чтение.

Рассмотрим уровень изоляции транзакции READ COMMITED

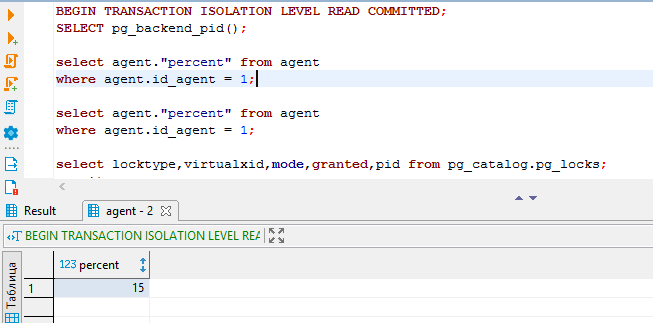
**Транзакция 1(pid = 2468)**

Рисунок 6: первый запрос данных

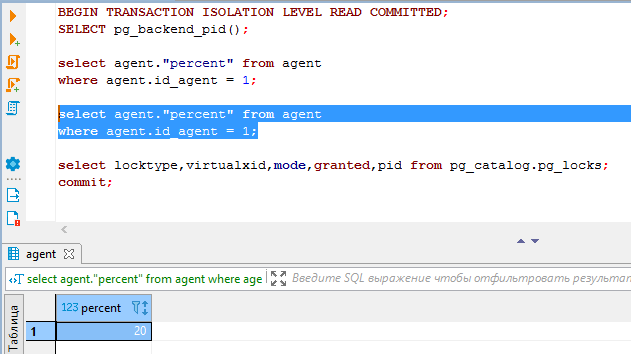


Рисунок 7 запрос данных после выполнения второй транзакции

**Транзакция 2(pid = 5420)**

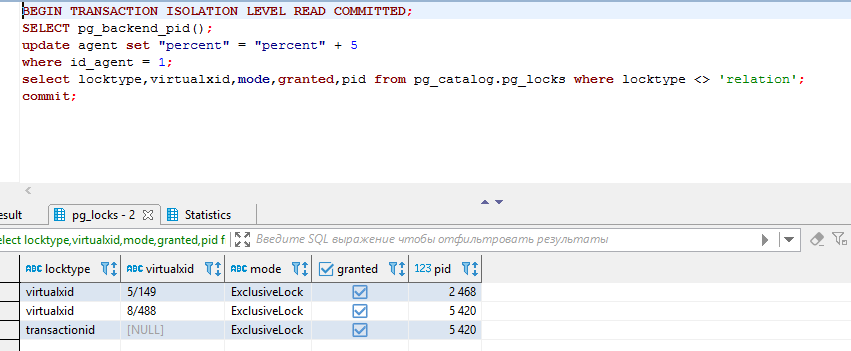


Рисунок 8: Транзакция 2 с запросом на обновление данных

**Блокировки транзакций**

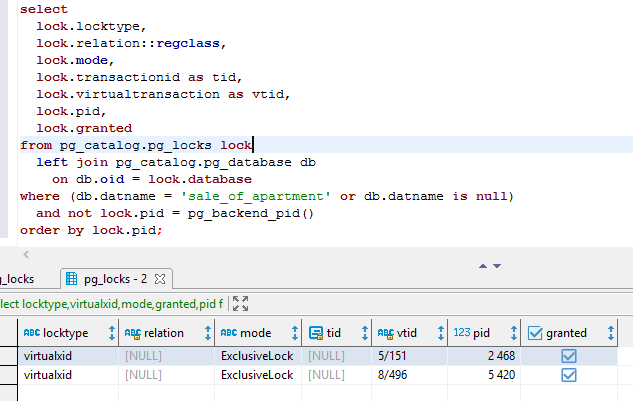


Рисунок 9: блокировки транзакций

Таким образом, транзакция 1 дважды выполняя один и тот же запрос, получает различные результаты.

Предупредить данную аномалию, без перехода на другой уровень изоляции можно следующими способами

* Использовать один SQL-оператор.

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

* Пользовательские блокировки.

Вручную установить исключительную блокировку или на все нужные строки. Это сводит на нет преимущества многоверсионности: вместо одновременного выполнения часть операций будет выполняться последовательно.

**Пользовательские блокировки**

Явно установим блокировку на столбец «процент».

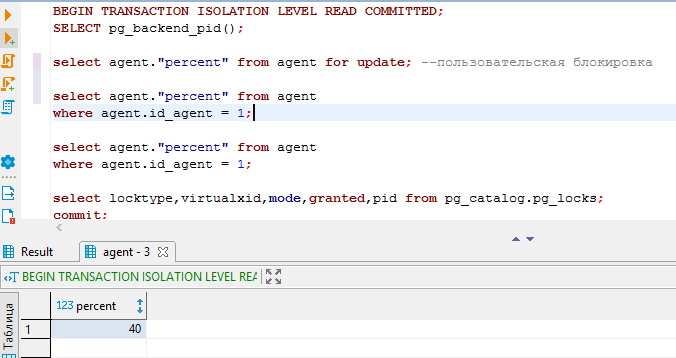
**Транзакция 1(pid = 2468)** 

Рисунок 10: первый запрос данных

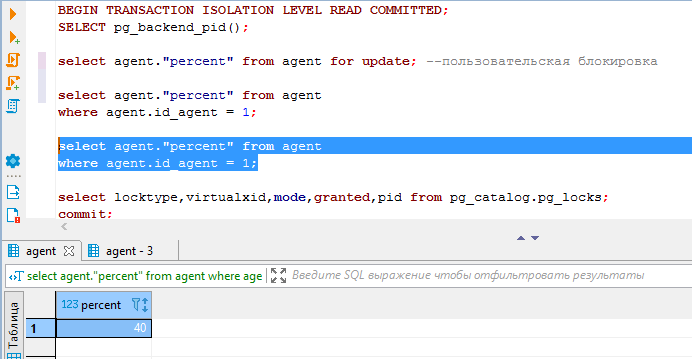


Рисунок 11: запрос данных после запуска второй транзакции

Как видно из результата повторного запроса аномалия неповторяемого чтения исчезла.

**Транзакция 2(pid = 5420)**

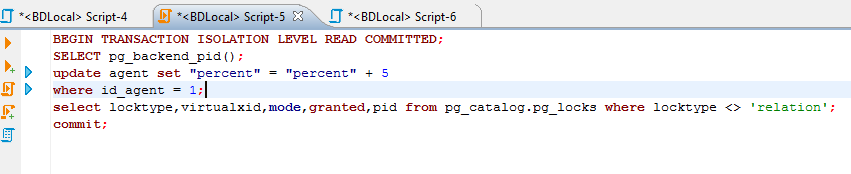


Рисунок 12: Транзакция 2 с запросом на обновление данных

**Блокировки транзакций**

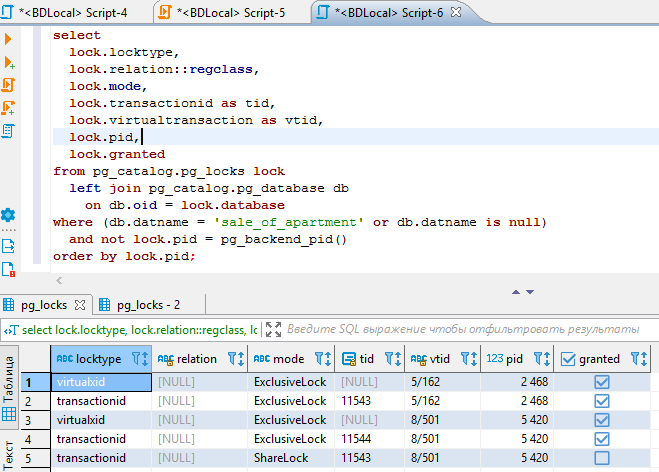


Рисунок 13: блокировки транзакций

Как видно из журнала блокировок транзакция 2 не получила требуемую блокировку и ожидает завершение 1й транзакции

**Повышение уровня изоляции транзакций до уровня REAPEATABLE READ.**

Аномалии неповторяемого чтения позволяет избежать повышение уровня изоляции транзакций до уровня REAPEATABLE READ или SERIALIZABLE

Рассмотрим уровень изоляции транзакций REPEATABLE READ

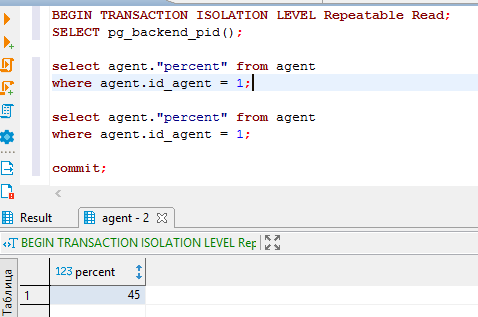
**Транзакция 1(pid = 6236)** 

Рисунок 14: первый запрос данных

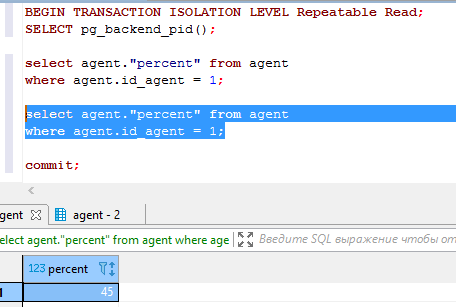


Рисунок 15: запрос данных после запуска второй транзакции

Как видно из результата повторного запроса, аномалия неповторяемого чтения исчезла.

**Транзакция 2(pid = 4260)**

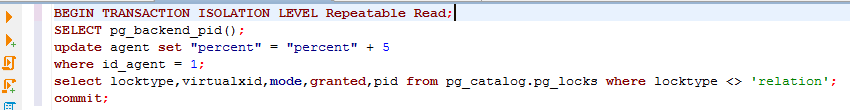


Рисунок 16: Транзакция 2 с запросом на обновление данных

**Блокировки транзакций**

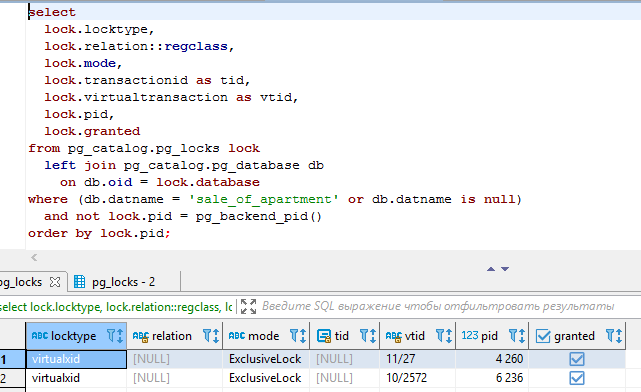


Рисунок 17: блокировки транзакций

* 1. **phantom rows — фантомные строки**

Смоделируем фантомные строки.

Рассмотрим уровень изоляции транзакции **READ COMMITTED**

**Транзакция 1(pid = 1372)**

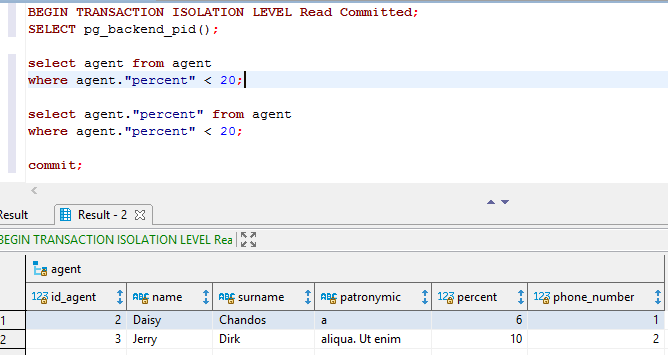


Рисунок 18: первый запрос данных

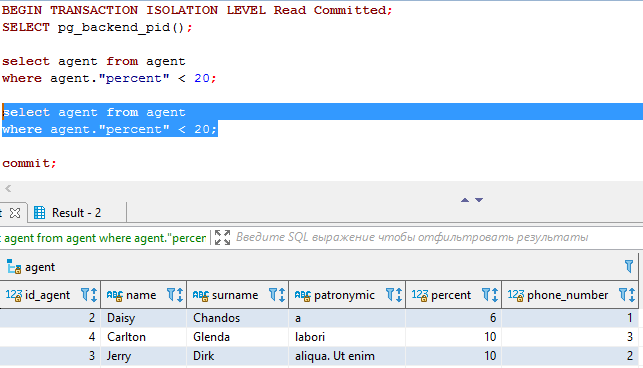


Рисунок 19: запрос данных после запуска второй транзакции

**Транзакция 2(pid = 9752)**

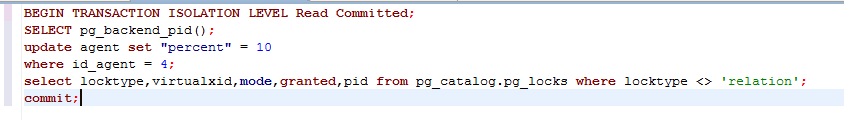


Рисунок 20: Транзакция 2 с запросом на обновление данных

**Блокировки транзакций**

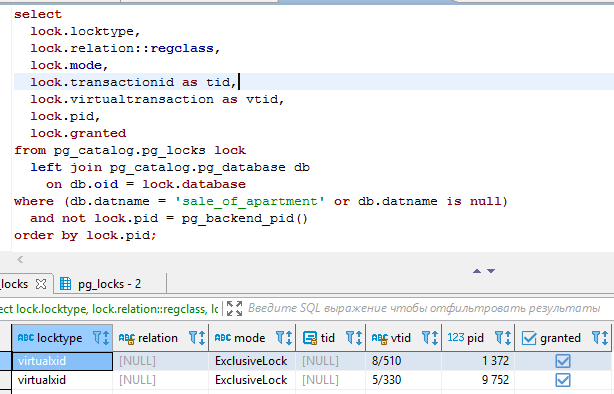


Рисунок 21: блокировки транзакций

**Повышение уровня изоляции транзакций до уровня Repeatable Read.**

Аномалии фантомного чтения позволяет избежать повышение уровня изоляции транзакций до уровня REAPEATABLE READ или SERIALIZABLE

Рассмотрим уровень изоляции транзакций REPEATABLE READ

**Транзакция 1(pid = 10220)**

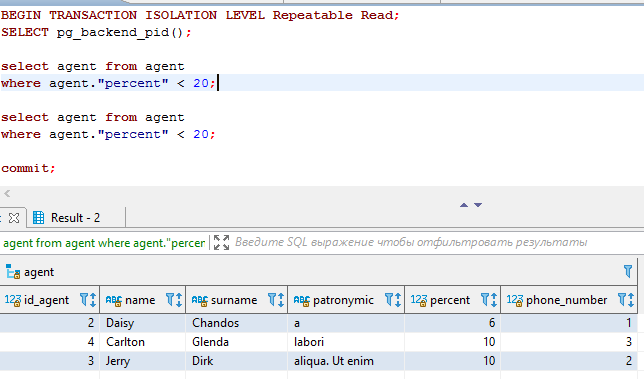


Рисунок 22: первый запрос данных

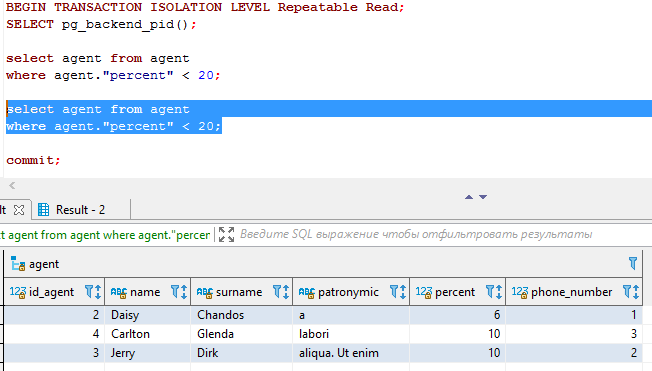


Рисунок 23: запрос данных после запуска второй транзакции

Как видно из результата повторного запроса, аномалия аномального чтения исчезла.

**Транзакция 2(pid = 7952)**

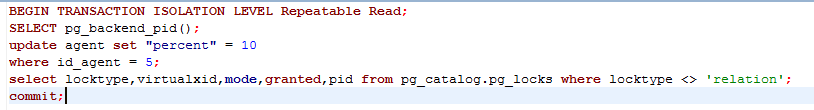


Рисунок 24: Транзакция 2 с запросом на обновление данных

**Блокировки транзакций**

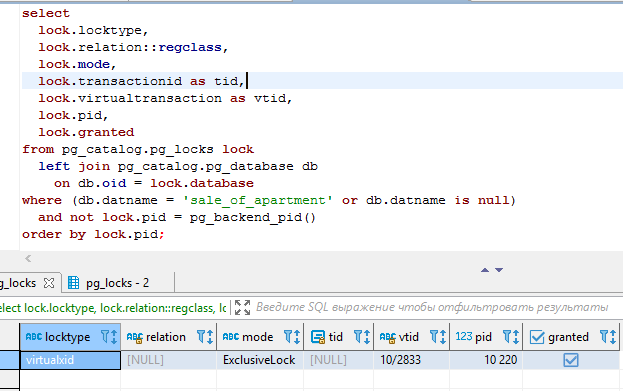


Рисунок 25: блокировки транзакций

1. **Взаимная блокировка/тупик (deadlock)**

Это ситуация, при которой, одному процессу для продолжения работы требуется ресурс, захваченный вторым процессом, а второму процессу требуется ресурс, захваченный первым процессом. В такой ситуации оба процесса оказываются в заблокированном состоянии и не могут продолжать работу.

Рассмотрим пример взаимной блокировки.

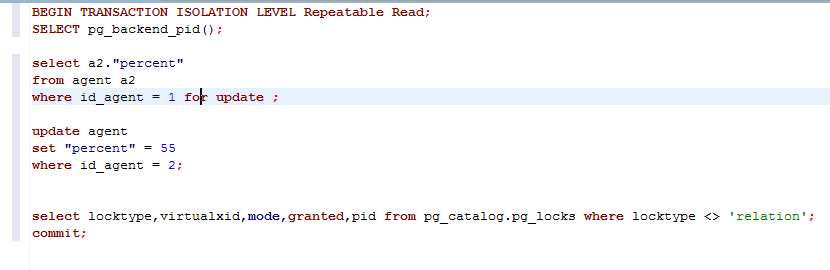


Рисунок 26 – Транзакция 1 для моделирования тупика

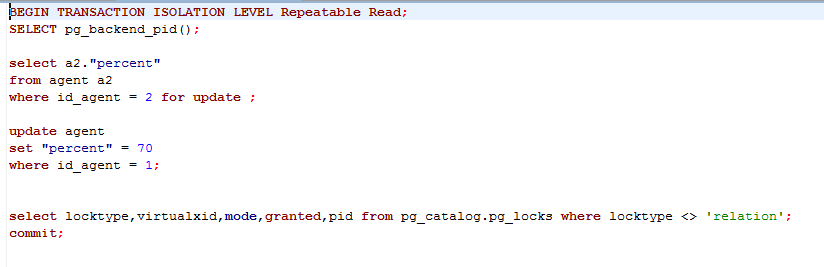


Рисунок 27 – Транзакция 2 для моделирования тупика

На операторе обновления мы получим блокировку в обеих транзакциях. Оператор SELECT в обеих транзакциях накладывает блокировку обновления на вторую и первую строки соответственно. Затем, при выполнении оператора UPDATE в транзакции 1, процесс пытается наложить эксклюзивную блокировку на первую строку, но не может этого сделать, т.к. она заблокирована транзакцией 2 и остается ждать его освобождения.

Аналогично транзакция 2 при выполнении оператора UPDATE пытается наложить эксклюзивную блокировку на вторую строку, но она уже заблокирована транзакцией 1 и процесс остается ждать освобождения блокировки.

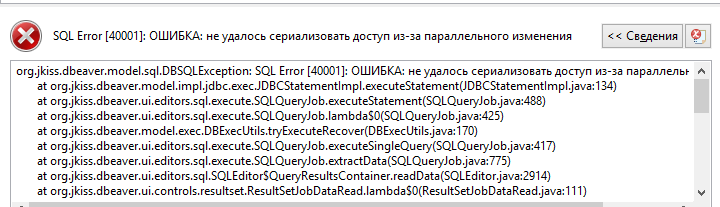


Рисунок 28 – Результат возникновения тупика

Обе транзакции ждут освобождения заблокированных друг другом ресурсов. Но на этом процессы не виснут. PostgreSQL с помощью встроенного менеджера блокировок определяет взаимные блокировки и разрешает их. Разрешает очень просто — жертвует одной из транзакций, т. е. попросту откатывает ее и возвращает ошибку. Вторая транзакция продолжит выполняться. Какая транзакция будет выбрана в качестве жертвы — определяет сам PostgreSQL.