Санкт-Петербургский государственный политехнический университет
Кафедра компьютерных систем и программных технологий

Отчёт по лабораторной работе

Дисциплина: Базы данных

Тема: Изучение механизма транзакций.

Выполнил студент гр. 43501/1 _____ А.О. Перешеин
(подпись)
Руководитель _____ А.В. Мяснов
(подпись)

“__” _____ 2015 г.

Санкт-Петербург
2015

1. Цель работы

Познакомить студентов с механизмом транзакций, возможностями ручного управления транзакциями, уровнями изоляции транзакций.

2. Программа работы

1. Изучить основные принципы работы транзакций.
2. Провести эксперименты по запуску, подтверждению и откату транзакций.
3. Разобраться с уровнями изоляции транзакций в Firebird.
4. Спланировать и провести эксперименты, показывающие основные возможности транзакций с различным уровнем изоляции.
5. Продемонстрировать результаты преподавателю, ответить на контрольные вопросы.

Работа проводится в IBExpert. Для проведения экспериментов параллельно запускается несколько сессий связи с БД, в каждой сессии настраивается уровень изоляции транзакций. Выполняются конкурентные операции чтения/изменения данных в различных сессиях, а том числе приводящие к конфликтам.

3. Выполнение работы

1) Транзакция – задача с определенными характеристиками, включающая любое количество различных операций (`SELECT`, `INSERT`, `UPDATE`, или `DELETE`) над данными в БД. Транзакция должна рассматриваться как единая операция над данными, т.е. или она была проведена полностью или она была не проведена вообще. Так как может возникнуть случай, когда во время выполнения транзакции, происходит ошибка, которая приводит к невозможности закончить текущую транзакцию, необходимы особые операции для подтверждения (commit) или отмены транзакции (rollback). Рассмотрим их на примере добавления записи в таблицу:

Подтверждение транзакции:

```
SQL> select * from artists;

    ART_ID ARTIST
========== ========================
         1 Pantera
         2 Amorphis
         3 Slayer
         4 Ice Cube
         5 Edge of Sanity

SQL> insert into artists values (6, 'Converge');
SQL> commit;
SQL> select * from artists;

    ART_ID ARTIST
========== ========================
         1 Pantera
         2 Amorphis
         3 Slayer
         4 Ice Cube
         5 Edge of Sanity
         6 Converge
```

В данном случае, после подтверждения транзакции изменения вступили в силу, т.е. была добавлена новая запись в таблицу.

Отмена транзакции:

```
SQL> select * from artists;

    ART_ID ARTIST
========== ========================
         1 Pantera
         2 Amorphis
         3 Slayer
         4 Ice Cube
         5 Edge of Sanity

SQL> insert into artists values (6, 'Converge');
SQL> rollback;
SQL> select * from artists;

    ART_ID ARTIST
========== ========================
         1 Pantera
         2 Amorphis
         3 Slayer
         4 Ice Cube
         5 Edge of Sanity
```

В этом случае транзакция (добавление записи) была отменена.

Также есть возможность устанавливать точки сохранения (savepoints), для возврата к определенному состоянию до начала транзакции:

```
SQL> select * from artists;

    ART_ID ARTIST
========== ========================
         1 Pantera
         2 Amorphis
         3 Slayer
         4 Ice Cube
         5 Edge of Sanity

SQL> savepoint sp1;
SQL> insert into artists values (6, 'Converge');
SQL> savepoint sp2;
SQL> insert into artists values (7, 'Bad Brains');
SQL> select * from artists;

    ART_ID ARTIST
========== ========================
         1 Pantera
         2 Amorphis
         3 Slayer
         4 Ice Cube
         5 Edge of Sanity
         6 Converge
         7 Bad Brains

SQL> rollback to sp2;
SQL> select * from artists;
```

```
    ART_ID ARTIST
=========== ========================
         1 Pantera
         2 Amorphis
         3 Slayer
         4 Ice Cube
         5 Edge of Sanity
         6 Converge

SQL> rollback to sp1;
SQL> select * from artists;

    ART_ID ARTIST
=========== ========================
         1 Pantera
         2 Amorphis
         3 Slayer
         4 Ice Cube
         5 Edge of Sanity
```

В этом случае мы последовательно вернулись к состоянию с 6 записями, а затем с 5 в таблице Artists, в которую в ходе транзакции добавили 2 записи.

2) Уровни изоляции – одна из характеристик транзакции, которая предписывает, как одна транзакция должна взаимодействовать с другими при доступе к одной и той же базе данных, с точки зрения видимости и блокировки. Уровень изолированности транзакции определяет, какие изменения, сделанные в других транзакциях, будут видны в данной транзакции. Каждая транзакция имеет свой уровень изоляции, который устанавливается при ее запуске и остается неизменным в течение всей ее жизни.

Уровень изоляции транзакции устанавливается с помощью команды SET TRANSACTION со следующим синтаксисом:

```
SET TRANSACTION
    [NAME hostvar]
    [READ WRITE | READ ONLY]
    [ [ISOLATION LEVEL] { SNAPSHOT [TABLE STABILITY]
                        | READ COMMITTED [[NO] RECORD_VERSION] } ]
    [WAIT | NO WAIT]
    [LOCK TIMEOUT seconds]
    [NO AUTO UNDO]
    [IGNORE LIMBO]
    [RESERVING <tables> | USING <dbhandles>]
```

Всего Firebird SQL предусматривает 3 уровня изоляции:

- READ COMMITTED («читать подтвержденные данные»)

  Уровень изоляции READ COMMITTED используется, когда мы хотим видеть все подтвержденные результаты параллельно выполняющихся (т. е. в рамках других транзакций) действий. Этот уровень изоляции гарантирует, что мы не сможем прочитать неподтвержденные данные, измененные в других транзакциях, и делает возможным прочитать подтвержденные данные.

  Для этого уровня изоляции возможно указание 2-х дополнительных параметров:

  RECORD_VERSION (при обращении к неподтвержденной версии записи возвращается ее старая версия)

  NO_RECORD_VERSION (при обращении к неподтвержденной версии записи выводится сообщение об ошибке - deadlock)
- SNAPSHOT («моментальный снимок»)

Этот уровень изоляции используется для создания "моментального" снимка базы данных. Все операции чтения данных, выполняемые в рамках транзакции с уровнем изоляции SNAPSHOT, будут видеть только состояние базы данных на момент начала запуска транзакции. Все изменения, сделанные в параллельных транзакциях, не видны в этой транзакции. В то же время SNAPSHOT не блокирует данные, которые он не изменяет.

- SNAPSHOT TABLE STABILITY

  Это уровень изоляции также создает "моментальный" снимок базы данных, но одновременно блокирует на запись данные, задействованные в операциях, выполняемые данной транзакцией. Это означает, что если транзакция SNAPSHOT TABLE STABILITY изменила данные в какой-нибудь таблице, то после этого данные в этой таблице уже не могут быть изменены в других параллельных транзакциях. Кроме того, транзакции с уровнем изоляции SNAPSHOT TABLE STABILITY не могут получить доступ к таблице, если данные в них уже изменяются в контексте других транзакций.

3) Проведем эксперименты с уровнями изоляции транзакций, для чего создадим 2 сессии связи с БД:

Рассмотрим уровень READ COMMITTED:

```
SQL> select * from artists;

    ART_ID ARTIST
========== ========================
         1 Pantera
         2 Amorphis
         3 Slayer
         4 Ice Cube
         5 Edge of Sanity
```

SQL> set transaction isolation level read committed; Commit current transaction (y/n)?y Committing. SQL> select * from artists; _ ART_ID ARTIST ========== ======================== 1 changed 2 Amorphis 3 Slayer 4 Ice Cube 5 Edge of Sanity	SQL> update artists set artist = 'changed' where art_id = 1; SQL> commit;

Таким образом, в ходе эксперимента выполняем неподтвержденное изменение данных таблицы Artists. При этом параллельно выполняется транзакция с уровнем изоляции READ COMMITTED, содержащая запрос select. Транзакция не выполняется, т.к.она обращается к неподтвержденным данным, но после подтверждения изменения мы увидим измененное содержимое таблицы.

Для этого же уровня изоляции на похожем примере рассмотрим влияние параметров RECORD_VERSION и NO RECORD_VERSION:

```
SQL> select * from artists;

    ART_ID ARTIST
```

```
============ ========================
         1 Pantera
         2 Amorphis
         3 Slayer
         4 Ice Cube
         5 Edge of Sanity
```

SQL> set transaction isolation level read committed record_version; Commit current transaction (y/n)?y Committing. SQL> select * from artists; ART_ID ARTIST ============ ======================== 1 Pantera 2 Amorphis 3 Slayer 4 Ice Cube 5 Edge of Sanity	SQL> update artists set artist = 'changed' where art_id = 1;

Для режима RECORD_VERSION при выполнении select действительно возвращается старая версия данных.

SQL> set transaction isolation level read committed no record_version; Commit current transaction (y/n)?y Committing. SQL> select * from artists; _ ART_ID ARTIST ============ ======================== 1 changed 2 Amorphis 3 Slayer 4 Ice Cube 5 Edge of Sanity	SQL> update artists set artist = 'changed' where art_id = 1; SQL> commit;

Для режима NO RECORD_VERSION сообщение ‘deadlock’ не выводится, но старые версии записей не выводятся.

Рассмотрим уровень SNAPSHOT:

```
SQL> select * from artists;

    ART_ID ARTIST
============ ========================
         1 changed
         2 Amorphis
         3 Slayer
         4 Ice Cube
         5 Edge of Sanity
```

SQL> set transaction isolation level snapshot; SQL> select * from artists; ART_ID ARTIST ============ ======================== 1 changed 2 Amorphis 3 Slayer	SQL> update artists set artist = 'Pantera' where art_id = 1; SQL> commit; SQL> select * from artists; ART_ID ARTIST ============ ======================== 1 Pantera 2 Amorphis 3 Slayer

4 Ice Cube 5 Edge of Sanity SQL> update artists set artist = 'Pantera' where art_id = 1; Statement failed, SQLSTATE = 40001 deadlock -update conflicts with concurrent update -concurrent transaction number is 1683	4 Ice Cube 5 Edge of Sanity

Видим, что транзакция, содержащая select, вывела «старое» содержимое, подтвержденное на момент старта транзакции. При таком уровне изоляции изменения (даже подтвержденные) данных из других транзакций не видны.
Также видно, что при изменении данных в рамках сделанного «моментального» снимка базы возникает конфликт между двумя транзакциями. В конечном итоге в силу вступили изменения из второй сессии (транзакция не с уровнем SNAPSHOT).

Рассмотрим уровень SNAPSHOT TABLE STABILITY:

SQL> select * from artists;

```
    ART_ID ARTIST
=========== ========================
         1 Pantera
         2 Amorphis
         3 Slayer
         4 Ice Cube
         5 Edge of Sanity
```

SQL> set transaction isolation level snapshot table stability Commit current transaction (y/n)?y Committing. SQL> update artists set artist = 'changed' where art_id = 1; SQL> commit; SQL> select * from artists; ART_ID ARTIST =========== ======================== 1 changed 2 Amorphis 3 Slayer 4 Ice Cube 5 Edge of Sanity	SQL> update artists set artist = 'changed' where art_id = 1; Statement failed, SQLSTATE = 40001 deadlock -update conflicts with concurrent update -concurrent transaction number is 1692

Также как и в предыдущем случае здесь возник конфликт между двумя транзакциями, но при этом транзакция с уровнем изоляции SNAPSHOT TABLE STABILITY запретила изменение таблицы из других транзакций, а не наоборот. И в этом случае в силу вступили изменения первой сессии (транзакция с уровнем SNAPSHOT TABLE STABILITY).

4. Выводы

В ходе выполнения работы было изучено создание и управление транзакциями, а также на примерах рассмотрены уровни изоляции. Все выполняемые транзакции должны удовлетворять следующим свойствам (ACID критерии):

- **Атомарность.** Выполнение по принципу "все или ничего".

- **Согласованность.** В результате транзакции система переходит из одного абстрактного корректного состояния в другое.
- **Изолированность.** Данные, находящиеся в несогласованном состоянии, не должны быть видны другим транзакциям, пока изменения не будут завершены
- **Долговечность.** Если транзакция зафиксирована, то ее результаты должны быть долговечными. Новые состояния всех объектов сохранятся даже в случае аппаратных или системных сбоев.

Транзакции позволяют выполнять контроль целостности данных при выполнении различных запросов, но если требовать последовательной обработки транзакций, то возможно уменьшение производительности, т.к. при параллельном выполнении транзакций возможны следующие проблемы:

- грязное чтение - чтение данных, добавленных или изменённых транзакцией, которая впоследствии не подтвердится (откатится);
- размытое чтение - при повторном чтении в рамках одной транзакции, ранее прочитанные данные оказываются изменёнными;
- фантомное чтение - одна транзакция в ходе своего выполнения несколько раз выбирает множество строк по одним и тем же критериям, другая транзакция в интервалах между этими выборками добавляет или удаляет строки или изменяет столбцы некоторых строк, используемых в критериях выборки первой транзакции, и успешно заканчивается; в результате получится, что одни и те же выборки в первой транзакции дают разные множества строк.

Для достижения компромисса между быстродействием существуют несколько уровней изоляции транзакций. При этом можно выделить соответствие между уровнями изоляции и возможными конфликтами:

Уровень изоляции	**Грязное чтение (Dirty Read)**	**Размытое чтение (Fuzzy Read)**	**Фантом (Phantom)**
Незафиксированное чтение (READ UNCOMMITTED)	возможно	возможно	возможно
Зафиксированное чтение (READ COMMITED)	невозможно	возможно	возможно
Повторяемое чтение (REPEATABLE READ)	невозможно	невозможно	возможно
Сериализуемость (SERIALIZABLE)	невозможно	невозможно	невозможно

Таким образом, получается, что сериализуемость – это способность к упорядочению параллельной обработки транзакций. Также можно установить соответствие между уровнями изоляции БД и уровнями изоляции транзакций в Firebird SQL:

Зафиксированное чтение	READ COMMITTED
Повторяемое чтение	SNAPSHOT
Сериализуемость	SNAPSHOT TABLE STABILITY