ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

ACID — набор требований к транзакционной системе, обеспечивающий наиболее надёжную и предсказуемую её работу — атомарность, согласованность, изоляцию, устойчивость.

GreenPlum — компания-разработчик массово-параллельной СУБД для хранилищ данных Greenplum на основе PostgreSQL.

MPP — класс архитектур параллельных вычислительных систем. Особенность архитектуры состоит в том, что память физически разделена.

Open-source —программное обеспечение с открытым исходным кодом. Исходный код таких программ доступен для просмотра, изучения и изменения.

PostgreSQL — свободная объектно-реляционная система управления базами данных.

СУБД — система управления базами данных.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Работа, проделанная мной в ходе технологической практики на предприятии ООО “ДатаБленд”, была посвящена изучению и формированию практических навыков работы с СУБД PostgreSQL.

В настоящее время PostgreSQL является одной из наиболее популярных СУБД, преимуществами которой является использование open-source подхода и наличие зрелых технических решений. Кроме того, на базе исходного кода PostgreSQL была разработана СУБД GreenPlum, использующая подход массовой параллельной обработки (MPP) для построения высоконагруженных витрин данных.

Цель эксплуатационной практики: овладеть теоретическими знаниями о транзакционной модели СУБД PostgreSQL и базовыми навыками управления транзакциями в ней.

Задачи практики:

- изучить архитектуру СУБД PostgreSQL;
- изучить концепции многоверсионности, изоляции и блокировок;
- Освоить базовые навыки управления транзакциями.

1 Общее устройство PostgreSQL

1.1 Клиент-серверный протокол

Взаимодействие с СУБД PostgreSQL осуществляется по клиент-серверному протоколу (рис. 1).

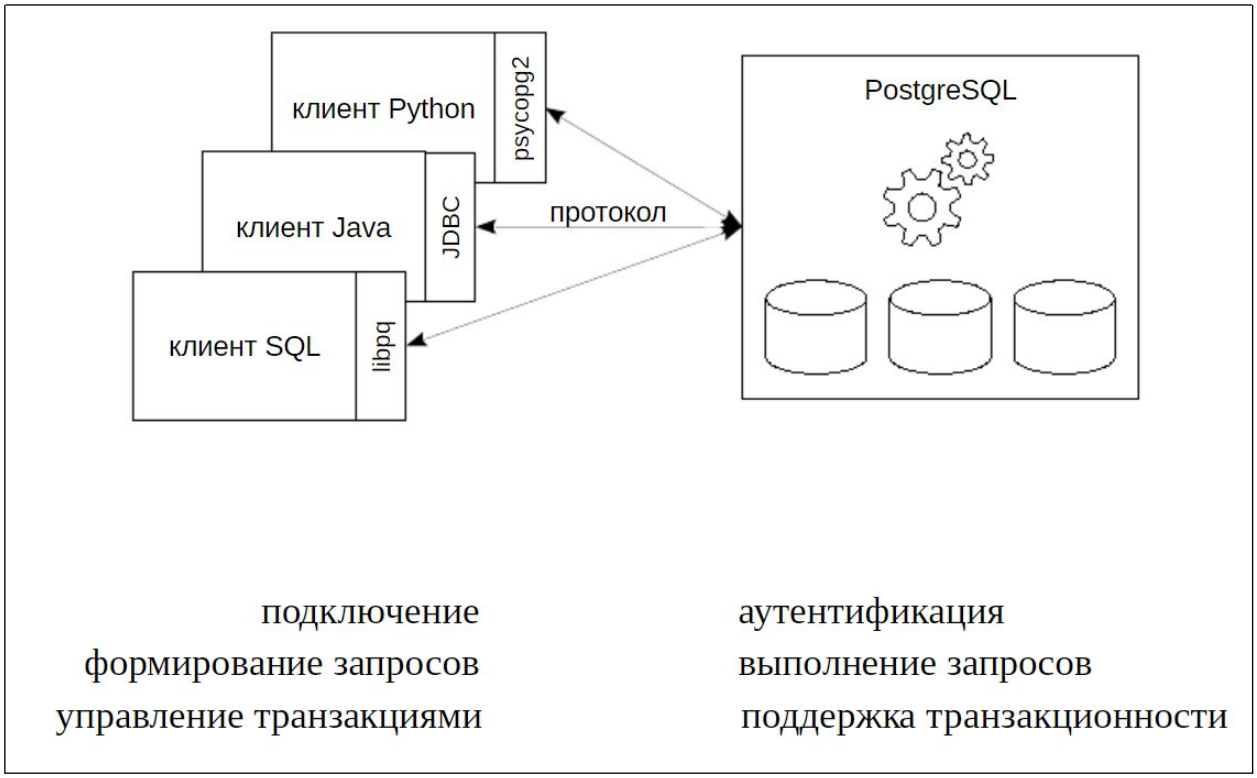

Рисунок 1 — клиент и сервер PostgreSQL

Клиентское приложение — например, psql или любая другая программа, написанная на любом языке программирования, — подключается к серверу и «общается» с ним [1]. Чтобы клиент и сервер понимали друг друга, они должны использовать один и тот же протокол взаимодействия. Обычно клиент использует драйвер, реализующий протокол и предоставляющий набор функций для использования в программе. Внутри драйвер может пользоваться стандартной реализацией протокола (библиотекой libpq [2]), либо реализовывать этот протокол самостоятельно.

Не так важно, на каком языке написан клиент — за разным синтаксисом будут стоять возможности, определенные протоколом. В большинстве заданий использован язык SQL с помощью клиента psql. В реальной жизни клиентскую часть на SQL пишут редко, но для учебных целей это удобно. Мы рассчитываем, что сопоставить команды SQL с аналогичными возможностями

какого-либо другого языка программирования не составит для вас большого труда. Протокол позволяет клиенту подключиться к одной из баз данных кластера. При этом сервер выполняет аутентификацию — решает, можно ли разрешить подключение, например, запросив пароль. Далее клиент посылает серверу запросы на языке SQL, а сервер выполняет их и возвращает результат. Наличие мощного и удобного языка запросов — одна из особенностей реляционных СУБД. Другая особенность — поддержка согласованной работы транзакций.

1.2 Транзакции

Последовательность работы с транзакцией и ее основные свойства проиллюстрированы на рисунке 2.

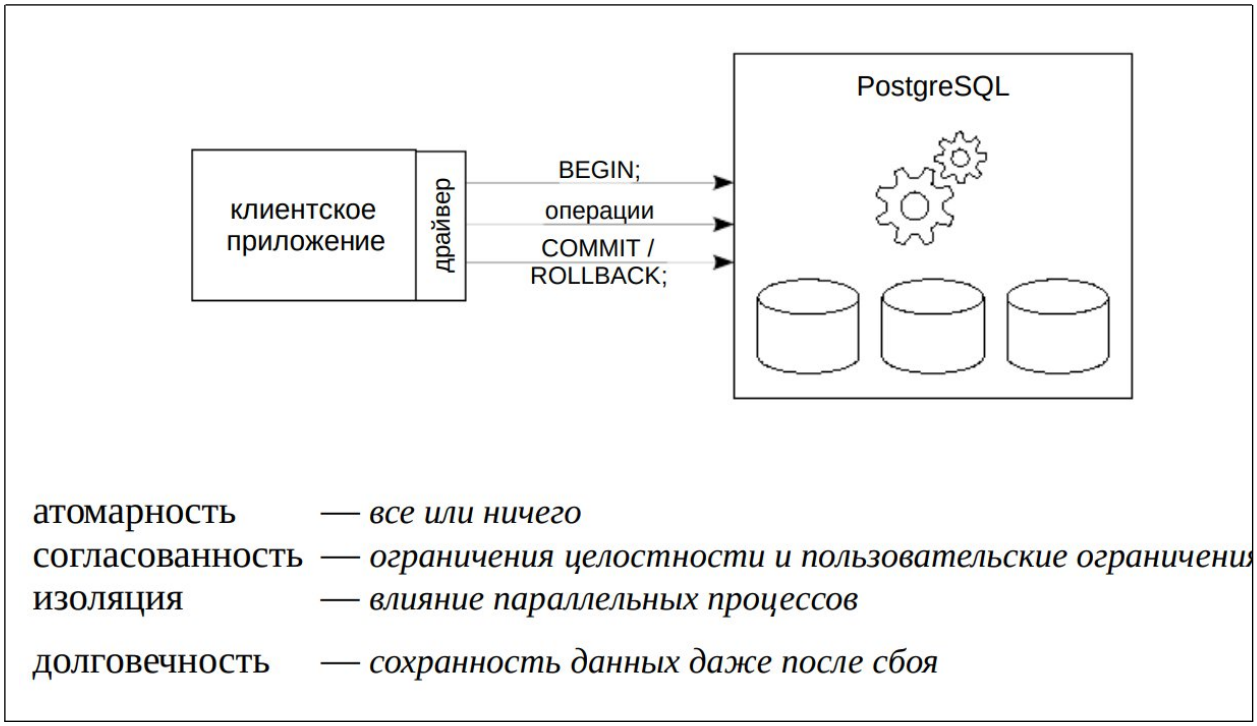

Рисунок 2 — свойства транзакции

Под транзакцией понимается последовательность операций, которая сохраняет согласованность данных при условии, что операции выполнены полностью и без помех со стороны других транзакций. От транзакций ожидают выполнения четырех свойств (ACID) [3].

Атомарность: транзакция либо выполняется полностью, либо не выполняется вовсе. Для этого начало транзакции отмечается командой

BEGIN, а конец — либо COMMIT (фиксация изменений), либо ROLLBACK (отмена изменений).

Согласованность: транзакция переводит базу данных из одного согласованного состояния в другое согласованное состояние (согласованность - соблюдение установленных ограничений).

Изоляция: другие транзакции, выполняющиеся одновременно с данной, не должны оказывать на нее влияния.

Долговечность: после того, как данные зафиксированы, они не должны потеряться даже в случае сбоя.

За управление транзакциями (то есть за определение команд, составляющих транзакцию, и за фиксацию или отмену транзакции) в PostgreSQL, как правило, отвечает клиентское приложение. На стороне сервера управлять транзакциями могут хранимые процедуры.

Ход выполнения практического задания по управлению транзакциями приведен в листинге 1.

Листинг 1 — управление транзакциями

```
- - По умолчанию psql работает в режиме автофиксации:

=> \echo :AUTOCOMMIT
on

- - Это приводит к тому, что любая одиночная команда, выданная без явного указания начала транзакции, сразу же фиксируется.

- - Создадим таблицу с одной строкой:

=> CREATE TABLE t(
  id integer,
  s text);
CREATE TABLE

=> INSERT INTO t(id, s) VALUES (1, 'foo');
INSERT 0 1

- - Строка будет видна и в другой транзакции

=> SELECT * FROM t;
```

```
 id |  s
----+-----
  1 | foo
(1 row)

- - Теперь попробуем начать транзакцию явно
=> BEGIN;
BEGIN

=> INSERT INTO t(id, s) VALUES (2, 'bar');
INSERT 0 1

- - Изменения не видны из другой транзакции
=> SELECT * FROM t;
 id |  s
----+-----
  1 | foo
(1 row)



- - Завершим транзакцию
=> COMMIT;
COMMIT

- - Теперь изменения видны
=> SELECT * FROM t;
 id |  s
----+-----
  1 | foo
  2 | bar
(2 rows)
```

2 Изоляция и многоверсионность

2.1 Многоверсионность

При одновременной работе нескольких сеансов возникает вопрос: что делать, если две транзакции одновременно обращаются к одной и той же строке? Если обе транзакции читающие, сложностей нет. Если обе пишущие — тоже (в этом случае они выстраиваются в очередь и выполняют изменения друг за другом). Самый интересный вариант — как взаимодействуют пишущая и читающая транзакции.

Простых пути два. Такие транзакции могут блокировать друг друга — но тогда страдает производительность. Либо читающая транзакция сразу может видеть изменения, сделанные пишущей транзакцией, даже если они не зафиксированы (это называется «грязным чтением») — но это очень плохо, ведь изменения могут быть отменены.

PostgreSQL идет сложным путем и использует многоверсионность — хранит несколько версий одной и той же строки (рис. 3). При этом пишущая транзакция работает со своей версией, а читающая видит свою [4].

Версии надо как-то отличать друг от друга. Для этого каждая из них хранит две отметки, определяющие «время действия» данной версии.

В качестве времени здесь используются всегда возрастающие номера транзакций (в действительности все немного сложнее, но это детали). Когда строка создается, она помечается номером транзакции, выполнившей команду INSERT. Когда удаляется — версия помечается номером транзакции, выполнившей DELETE (но физически не удаляется). UPDATE состоит из двух операций DELETE и INSERT.

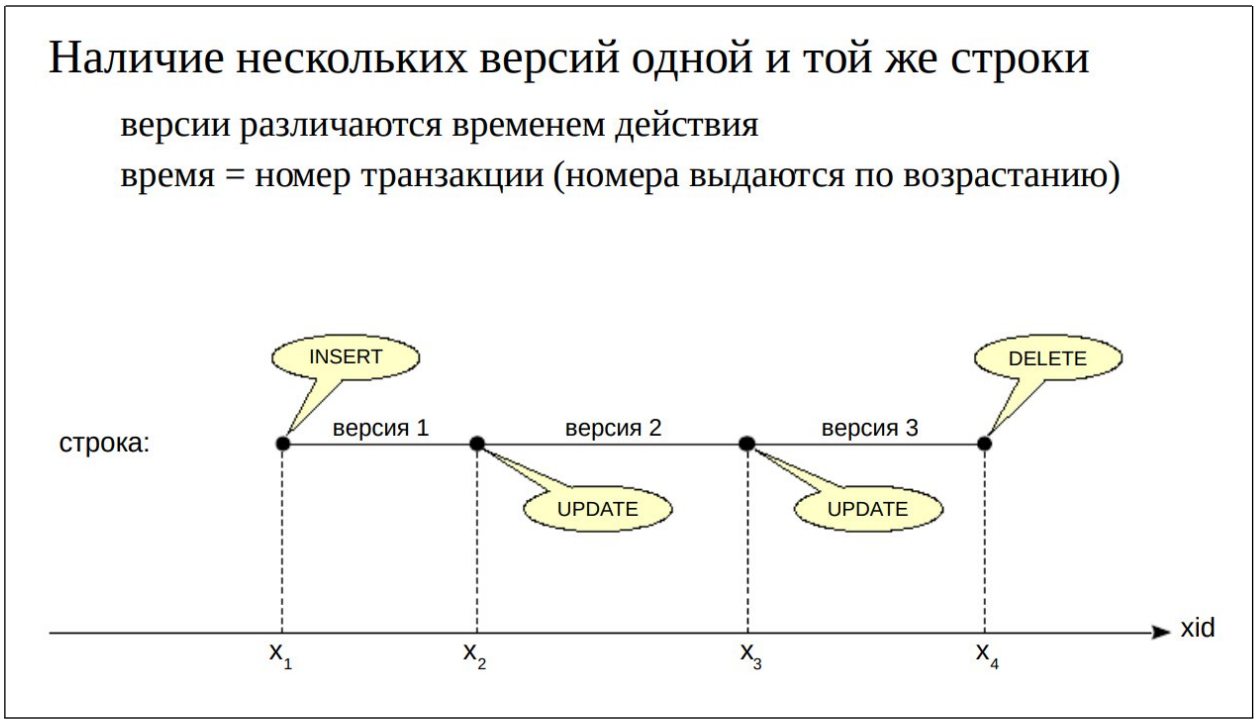

Рисунок 3 — многоверсионность

2.2 Уровни изоляции

Стандарт SQL определяет четыре уровня изоляции: чем строже уровень, тем меньше влияния оказывают параллельно работающие транзакции друг на друга [5]. Во времена, когда стандарт принимался, считалось, что чем строже уровень, тем сложнее его реализовать и тем сильнее его влияние на производительность (с тех пор эти представления несколько изменились).

Самый нестрогий уровень Read Uncommitted допускает грязные чтения. Он не поддерживается PostgreSQL, поскольку не представляет практической ценности и не дает выигрыша в производительности.

Уровень Read Committed является уровнем изоляции по умолчанию в PostgreSQL. На этом уровне снимки данных строятся в начале выполнения каждого оператора SQL. Таким образом, оператор работает с неизменной и согласованной картиной данных, но два одинаковых запроса, следующих один за другим, могут показать разные данные.

На уровне Repeatable Read снимок строится в начале транзакции (при выполнении первого оператора) — поэтому все запросы в одной транзакции видят одни и те же данные. Этот уровень удобен, например, для отчетов, состоящих из нескольких запросов.

Уровень Serializable гарантирует полную изоляцию: можно писать операторы так, как будто транзакция работает одна. Но при этом некоторая доля транзакций будет прерываться; приложение должно уметь повторять такие транзакции.

Ход выполнения практического задания по работе с версиями строк приведен в листинге 2.

Листинг 2 — работа с версиями строк

```
- - Создадим таблицу:

=> CREATE TABLE t(s text);
CREATE TABLE


- - И вставим одну строку. Если не начать транзакцию явно командой BEGIN, psql выполняет
```

команду и немедленно фиксирует результат:

```
=> INSERT INTO t VALUES ('Первая версия');
INSERT 0 1
```

- - Начнем транзакцию и выведем ее номер:

```
=> BEGIN;
BEGIN

=> SELECT pg_current_xact_id();
 pg_current_xact_id
--------------------
                736
(1 row)
```

- - Транзакция видит первую (и пока единственную) версию строки:

```
=> SELECT *, xmin, xmax FROM t;
      s       | xmin | xmax
---------------+------+------
 Первая версия |  735 |    0
(1 row)
```

- - Здесь скрытые столбцы показывают номера транзакций, ограничивающих видимость версии строки: xmin — номер предыдущей транзакции, которая создала версию, а xmax=0 означает, что эта версия актуальна.

- - Теперь начнем другую транзакцию в другом сеансе:

```
=> BEGIN;
BEGIN

=> SELECT pg_current_xact_id();
 pg_current_xact_id
--------------------
                737
(1 row)
```

- - Транзакция видит ту же единственную версию:

```
=> SELECT *, xmin, xmax FROM t;
      s       | xmin | xmax
---------------+------+------
 Первая версия |  735 |    0
(1 row)
```

- - Теперь изменим строку во второй транзакции.

```
=> UPDATE t SET s = 'Вторая версия';
UPDATE 1
```

- - Вот что получилось:

```
=> SELECT *, xmin, xmax FROM t;
      s       | xmin | xmax
---------------+------+------
 Вторая версия |  737 |    0
(1 row)
```

- - Поскольку изменение не зафиксировано, первая транзакция продолжает видеть первую версию строки.

```
=> SELECT *, xmin, xmax FROM t;
      s       | xmin | xmax
---------------+------+------
 Первая версия |  735 |  737
(1 row)
```

- - xmax — значение показывает, что в настоящий момент другая транзакция меняет строку. Вообще говоря, такое «подглядывание» нарушает изоляцию, поэтому поля xmin и xmax скрыты и в реальной работе их использовать не стоит.

- - Теперь зафиксируем изменения.

```
=> COMMIT;
COMMIT
```

- - Теперь и первая транзакция видит вторую версию строки.

```
=> SELECT *, xmin, xmax FROM t;
      s       | xmin | xmax
---------------+------+------
 Вторая версия |  737 |    0
(1 row)
```

- - После фиксации первая версия строки больше не будет видна ни в одной транзакции.

```
=> COMMIT;
COMMIT
```

2.3 Блокировки

Что же дает многоверсионность? Она позволяет обойтись необходимым минимумом блокировок, тем самым увеличивая производительность системы.

Основные блокировки устанавливаются на уровне строк. При этом чтение никогда не блокирует ни читающие, ни пишущие транзакции. Изменение строки не блокирует ее чтение. Единственный случай, когда транзакция будет ждать освобождения блокировки — если она пытается менять строку, которая уже изменена другой, еще не зафиксированной транзакцией.

Блокировки также устанавливаются на более высоком уровне, в частности, на таблицах. Они нужны для того, чтобы никто не смог удалить таблицу, пока другие транзакции читают из нее данные, или чтобы запретить доступ к перестраиваемой таблице. На практике удаление и перестроение таблиц выполняются редко, поэтому обычно не вызывают проблем. Однако нужно учитывать, что перестроение таблицы происходит неявно при некоторых изменениях ее структуры и полностью блокирует доступ к таблице и ее индексам.

Все необходимые блокировки устанавливаются автоматически и автоматически же снимаются при завершении транзакции. Можно также установить и дополнительные пользовательские блокировки; необходимость в этом возникает не часто.

Ход выполнения практического задания на работу с блокировками приведен в листинге 3.

Листинг 3 — работа с блокировками

```
- - пусть обе транзакции попытаются изменить одну и ту же строку.

=> BEGIN;
BEGIN

=> UPDATE t SET s = 'Третья версия' RETURNING *;
     s
---------------
```

```
 Третья версия
(1 row)

UPDATE 1
```

- - И во второй транзакции:

```
=> BEGIN;
BEGIN

=> UPDATE t SET s = 'Четвертая версия' RETURNING *;
```

- - Вторая транзакция «повисла»: она не может изменить строку, пока первая транзакция не снимет блокировку.

```
=> COMMIT;
COMMIT
```

- - Теперь вторая транзакция может продолжить выполнение:

```
       s
------------------
 Четвертая версия
(1 row)

UPDATE 1

=> COMMIT;
COMMIT
```

- - Обе транзакции зафиксировали свои изменения. Первый сеанс снова читает таблицу и видит актуальную строку — это результат, зафиксированный второй транзакцией:

```
=> SELECT * FROM t;
       s
------------------
 Четвертая версия
(1 row)
```

ЗАКЛЮЧЕНИЕ

Во время прохождения практики была изучена архитектура и транзакционная модель СУБД PostgreSQL, концепции многоверсионности, изоляции и блокировок.

Кроме того, были получены практические управления транзакциями в PostgreSQL.

Результаты проделанной работы представлены в настоящем отчете, составленном в соответствии с требованиями государственного стандарта [6].

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. PostgreSQL. DBA1 : сайт. – URL: https://postgrespro.ru/education/courses/DBA1 (дата обращения: 01.07.2024)
2. Libpq docs : сайт. – URL: https://www.postgresql.org/docs/current/libpq.html (дата обращения: 03.07.2024)
3. Требования ACID на простом языке // Habr : сайт. – URL: https://habr.com/ru/articles/555920/ (дата обращения: 04.07.2024)
4. Изоляция и многоверсионность в Postgresql // Sysadminium : сайт. – URL: https://sysadminium.ru/izolyaciya_i_mnogoversionnost_v_postgresql/ (дата обращения: 08.07.2024)
5. Transaction Isolation Levels in DBMS // GeeksForGeeks : сайт. – URL: https://www.geeksforgeeks.org/transaction-isolation-levels-dbms/ (дата обращения: 11.07.2024)
6. ГОСТ 7.32 – 2017. Отчет о научно-исследовательской работе. – Москва: Стандартинформ, 2017. – 32 с.