Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем

Семенов Александр Сергеевич

Библиотека транзакционного доступа к файлам из PostgreSQL

Производственная практика

Научный руководитель:
доц. каф. СП, к.ф.-м.н. Луцив Д.В.

Санкт-Петербург
2024

Оглавление

Введение

Сегодня сложно представить себе организацию, работа которой обходится без информационных систем. Одной из основных составляющих любой информационной системы являются данные. Для их хранения и управления используются различные технологии, например, разнообразные файловые хранилища и в частном случае базы данных. Для настраивания и администрирования баз данных применяется набор программных средств, называемый системой управления базами данных (СУБД).

Одной из самых популярных объектно-реляционных СУБД является PostgreSQL, которая широко используется в различных областях, включая финансы, науку и образование. Она имеет открытый исходный код, поддерживает репликации, хранимые процедуры и прочие полезные функции, а также совместима с набором требований ACID. Одним из требований ACID является атомарность, которая гарантирует, что транзакция либо будет выполнена полностью, либо не выполняется совсем. Это позволяет сохранять согласованность данных в базе, что может быть критично для многих информационных систем, например, для банковских или систем, взаимодействующих с рынком ценных бумаг.

Для хранения данных внутри базы в PostgreSQL предусмотрено множество типов, таких как integer, text, boolean и прочие. Эти типы данных позволяют эффективно хранить и обрабатывать структурированные данные, однако они не подходят для неструктурированных данных, например, бинарных. Основное отличие этих типов данных в том, что неструктурированные данные не соответствуют заранее определенной структуре, из-за этого подходы к хранению и обработке такого рода информации должны отличаться от соответствующих подходов работы со структурированными данными. Примерами бинарных данных, которые может быть полезно хранить информационной системе, могут быть pdf-документы, картинки, архивы, электронные письма и тп.

Если бинарные данные имеют небольшой размер, то часто для их

хранения помимо основной таблицы заводят вспомогательную, в которой хранят мета-данные, такие как дата последнего изменения и ссылка на конкретную версию файла. Такой способ вполне сгодится, если планируется хранить файлы небольшого размера, которые помещаются в страницу фиксированного размера (в PostgreSQL обычно 8 Кб). Проблемы возникают, когда появляется необходимость хранить большие файлы, так как это будет сказываться на производительности базы данных.

В данной работе будут рассмотрены подходы к транзакционному доступу к бинарным файлам внутри СУБД PostgreSQL, предложен подход, позволяющий делать это эффективно с точки зрения занимаемого дискового пространства и времени, а также поддерживающий транзакционность операций над бинарными данными.

1. Постановка задачи

Данная работа выполняется в рамках проекта, над которым работают два человека. Целью данной работы является написание библиотеки для транзакционного доступа к бинарным данным, которая будет использоваться для написания расширения для PostgreSQL, позволяющего управлять данными при помощи синтаксиса SQL.

Для достижения цели были поставлены следующие задачи:

- Изучить существующие подходы к решению задачи хранения бинарных данных в PostgreSQL.
- Определить подход к хранению бинарных данных.
- Реализовать библиотеку для написания расширения PostgreSQL для транзакционного доступа к бинарным данным.
- Провести тестирование полученного решения.

2. Обзор

Требования ACID[1] к транзакционной системе гарантируют, что транзакции будут выполняться целиком, система в любой момент времени находится в согласованном состоянии, несколько работающих одновременно транзакций не влияют на результат других, а также выполняется условие надежности, при котором можно быть уверенным, что данные не пропадут, если было получено соответствующее подтверждение об успешной фиксации транзакции.

Выполнение этих требований для хранения данных, в том числе и бинарных, дает некоторые преимущества, например, одновременный доступ к данным из нескольких потоков не приводит к конфликтам [6]. Это может быть критично для некоторых систем, которым необходимо работать с неструктурированными бинарными форматами файлов. Далее в обзоре будут рассматриваться подходы к решению поставленной задачи, которые могут удовлетворять требованиям ACID.

2.1. Стандартные средства PostgreSQL

В СУБД PostgreSQL для хранения бинарных данных предусмотрены типы bytea и BLOB[2] [4]. Тип bytea является двоичной строкой переменной длины. Этот тип использует механизм TOAST[3], который позволяет разбивать большие файлы на множество мелких. Для этого заводится специальная таблица, которая хранит ссылки на части файла и с помощью которой впоследствии можно восстановить файл из частей [10]. Однако этот механизм не подходит для поддержки обновлений данных, т.к. в случае обновления данных sql-операцией UPDATE, механизм TOAST будет дублировать все части сохраненных данных, даже если изменения затрагивают какую-то одну часть из них, из-за чего производительность на больших данных будет низкой [7].

Помимо bytea, в PostgreSQL есть средство хранения больших объ-

[1] Atomicity, Consistency, Isolation, Durability

[2] Binary Large Object

[3] The Oversized Attribute Storage Technique

ектов BLOB, которое обеспечивает потоковый доступ к данным, однако для работы с ним нужно использовать отличающийся от стандартного интерфейс. Также у этого способа хранения есть ограничение на размер файла — 2 Гб, из-за чего такой способ не подходит для больших бинарных данных.

Несмотря на то, что стандартные способы хранения бинарных данных в PostgreSQL поддерживают транзакционность, они не позволяют эффективно работать с бинарными файлами больших размеров. В виду того, что PostgreSQL обрабатывает данные в рамках транзакций, работа с очень большими файлами может привести к длительным блокировкам и занять значительное количество времени и ресурсов, что неэффективно для транзакционной системы. Также большие объекты замедляют процессы резервного копирования и восстановления.

2.2. Система контроля версий Git

Система контроля версий Git позволяет хранить историю изменений файлов, поддерживает ветвления и работу с конфликтами. Наибольшую популярность эта система обрела среди разработчиков, потому что позволяет эффективно работать с изменением текстовых файлов с исходными кодами программ.

Помимо текстовых, Git позволяет фиксировать изменения и бинарных файлов. Поэтому с помощью этой системы можно организовать транзакционный доступ к файлам на внешнем файловом хранилище, используя фиксирование изменений файлов при помощи коммитов — снапшотов всего состояния репозитория. Можно было бы представить себе решение поставленной задачи следующим образом: завести внешнее хранилище данных, настроить там Git и подключить его к СУБД.

В проекте Pylons[4] есть разработка прослойки для файловой системы, которая использует Git для поддержки ACID гарантий [1]. Как отмечают авторы, идея разработки такого инструмента появилась из-за удобства в небольших проектах напрямую писать и читать из файловой

[4] `https://pylonsproject.org`

системы. Для таких проектов может быть излишним взаимодействие с базами данных, что может затруднять развитие проекта на ранних этапах. В результате была разработана библиотека с API на языке Python, которая обеспечивает согласованную запись и чтение несколькими потоками, а также умеет объединять одновременные модификации одного и того же файла. Кроме того, при использованием этой библиотеки остается доступным полная история всех изменений, так как для функционирования используется резервный Git-репозиторий.

Однако при использовании подобного рода решений возникает существенный недостаток: Git сжимает и сохраняет полные версии двоичных файлов, что не оптимально, если бинарных файлов много и они большого размера. Из-за этого любое изменение файла, даже самое небольшое, будет приводить к созданию копии файла, а если мы имеем дело с большими файлами, то размер файлового хранилища будет существенно расти [9]. Также переключение между версиями файлов будет занимать существенное время.

2.3. Прочие подходы

Далее будут рассмотрены другие подходы к организации транзакционного хранилища, которые могут применяться к решению поставленной задачи. Например, авторы статьи [8] решают проблему повсеместного использования электронных таблиц во внутренних процессах в автомобильном секторе бизнеса. Они отмечают, что существующий подход имеет значительные недостатки, например, отсутствие проверки ограничения и поддержки одновременной работы нескольких пользователей. Для решения этой проблемы они предлагают концепцию умных файлов, которые совмещают в себе преимущества СУБД и системы управления рабочими процессами. Умные файлы являются исполняемыми и сами контролируют права доступа и свою согласованность. Внутри такого файла есть своя файловая система, которая даёт возможность протоколировать все операции и восстанавливать файлы в случае ошибки, а также поддерживает механизм версионирования. Так-

же умные файлы учитывают, что пользователь может долго работать над одним файлом, из-за чего транзакция может растянутся на длительный промежуток времени, чего обычно не происходит при работе с СУБД. Это реализовано при помощи отложенных транзакций, сохраняемых локально у пользователя. Когда пользователь фиксирует изменения, то все отложенные транзакции фиксируются, и только после этого другие пользователя увидят изменения, внесенные в этот файл. Концепция умных файлов представлена на Рис.1.

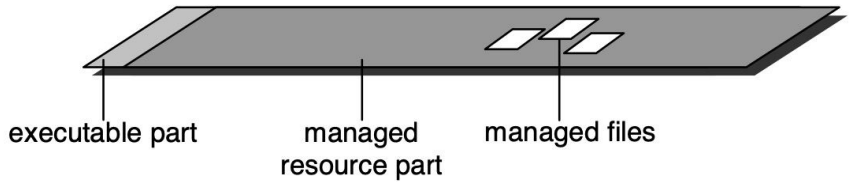

Рис. 1: Схема умного файла

Теоретически, концепцию умных файлов можно распространить не только на электронные таблицы, но и на прочие файлы бинарного формата. Однако несмотря на все преимущества такого подхода, его нельзя в полной мере назвать подходящим для решения поставленной задачи в виду того, что такой подход, как минимум, требует пересмотра процессов работы с файлами в компаниях, то есть переобучение людей, а также размер таких файлов будет больше из-за того, что каждый файл будет содержать в себе служебные данные для поддержки всех перечисленных функций. В данной работе предлагается механизм хранения бинарных файлов в СУБД, который лишен этих недостатков.

Другая публикация [5], которую интересно рассмотреть в рамках данной работы, написана коллективом ученых из университета Беркли, Стэнфордского университета и компании Databricks. Они представляют Delta Lake — слой хранения данных с открытым исходным кодом, который поддерживает ACID гарантии, оптимизирован для потоковой работы с большими объёмами данных, а также имеет интеграцию с многими фреймворками для распределённой обработки неструктурирован-

ных данных, например, Apache Spark. Авторы отмечают, что существующие способы облачного хранения объектов не удобны при большой потоковой обработке из-за возникающих проблем с согласованностью данных и низкой производительностью, а также отмечают, что доступ к API таких сервисов может быть очень дорогим.

Delta Lake использует формат файлов Parquet[5] для хранения данных. Это столбцовый формат хранения, оптимизированный для аналитических запросов. Помимо этого, в Delta Lake используется транзакционный журнал, который отслеживает изменения файлов и благодаря которому поддерживаются требования ACID. Сам журнал является набором объектов JSON, которые содержат массив последовательных действий, которые необходимо выполнить, чтобы получить следующую версию таблицы. Примерная схема хранения объектов в Delta Lake представлена на Рис.2.

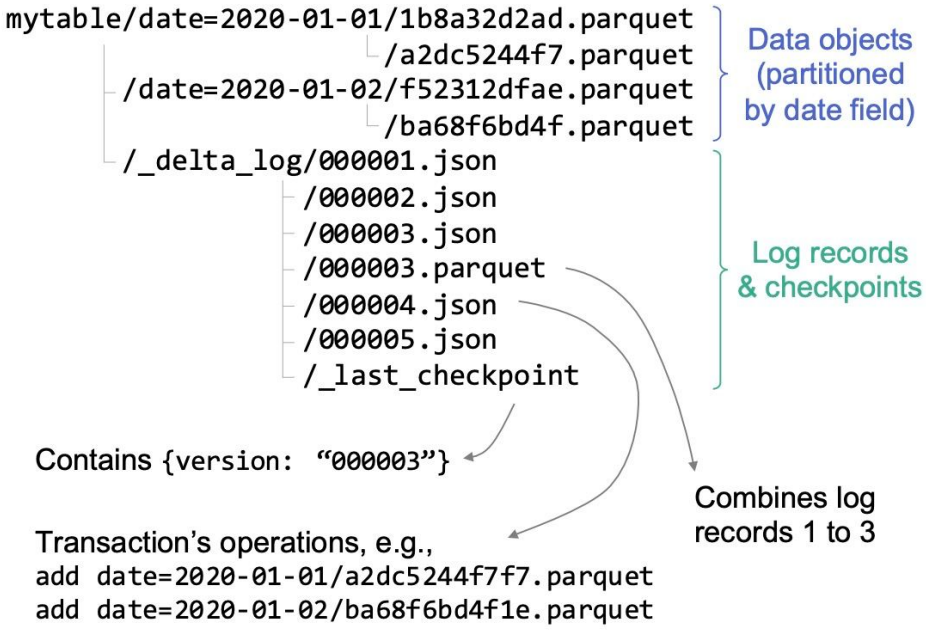

Рис. 2: Пример хранения объектов в Delta Lake

Однако несмотря на то, что Delta Lake имеет необходимую в контексте данной работы поддержку ACID гарантий и множество фрейм-

[5] https://parquet.apache.org

ворков[6], в том числе и для языка C++, данный метод не подходит для решения поставленной задачи в виду того, что данные хранятся в специализированном формате Parquet, который хоть и поддерживает бинарные форматы данных, однако не эффективен при работе с большими бинарными файлами. К тому же преобразование бинарных файлов в формат, понятный Parquet, привнесет дополнительную задержку по времени.

[6] `https://delta.io/integrations/`

3. Предлагаемое решение

Существуют два основных способа решения поставленной задачи: написание расширения для PostgreSQL, которое реализовывало бы логику версионирования бинарных файлов внутри базы данных, или хранение файлов в другом месте, например, в файловом хранилище. Первый вариант предполагает, что внутри СУБД файлы хранятся в одном из типов bytea или BLOB, однако операции с такими данными были бы медленнее, чем если бы файлы хранились в файловом хранилище [2]. С другой стороны, хранение файлов в файловой системе не гарантирует атомарность операций над ними.

3.1. Описание решения

Поставленную задачу можно решить, совместив два подхода. В данной работе предлагается решить проблему хранения и транзакционного доступа к бинарным данным в СУБД PostgreSQL следующим образом: заводится файловое хранилище с файловой системой, которая позволяет на её основе реализовать необходимую для транзакционного доступа к файлам функциональность. Предполагается, что у сервера базы данных есть доступ к этому хранилищу. На основе файловой системы будет реализована библиотека, позволяющая организовать транзакционный доступ к файлам.

Используя эту библиотеку, будет реализовано расширение для СУБД PostgreSQL, позволяющее взаимодействовать с файлами через стандартный синтаксис SQL, то есть поддерживающий стандартные операции SELECT, INSERT, UPDATE и DELETE, а также позволяющее сохранять, атомарно получать доступ и версионировать бинарные файлы из СУБД. Благодаря этому будет возможность взаимодействовать с файлами стандартными средствами СУБД, инкапсулируя от конечного пользователя информацию о том, как и где эти файлы на самом деле хранятся.

3.2. Выбор файловой системы

Были выделены основные критерии отбора файловой системы для написания библиотеки транзакционного доступа к файлам для последующего написания расширения для PostgreSQL:

- Наличие библиотеки для взаимодействия с файловой системой из программного кода;
- Поддержка транзакций через механизм copy-on-write;
- Поддержка создания снапшотов для реализации версионирования файлов за константное время.

Снапшоты — это специальные снимки, которые хранят полное состояние диска или файла в определенный момент времени. Благодаря этому механизму появляется возможность хранить несколько снимков определенного файла или директории и возвращаться к любому снимку в случае необходимости.

Механизм copy-on-write позволяет при изменении данных не менять их напрямую, а создает копию данных в другом месте и изменяет её. Это позволяет избежать ошибок в данных при неудачных изменениях. Также эта технология позволяет увеличить скорость записи, так как она стирает старые данные только когда появляется в этом необходимость.

Выбор происходил между файловыми системами EXT4, BTRFS и ZFS, так как эти файловые системы чаще всего используются для сетевых хранилищ, а также у всех них есть библиотеки для взаимодействия с ними.

Несмотря на то, что EXT4 является самой стабильной из рассматриваемых файловых систем, а также по умолчанию используется на большинстве операционных систем семейства Unix, она не поддерживает механизм copy-on-write, поэтому дальше эта файловая система рассматриваться не будет.

Файловые системы BTRFS и ZFS обе поддерживают механизм copy-on-write и создание снапшотов. Однако, при работе с большими объё-

мами данных, BTRFS показывает более низкую эффективность в сравнении с ZFS [11].

Поэтому в дальнейшем для решения поставленной задачи будет использоваться файловая система ZFS. Это хороший выбор для хранения и версионирования бинарных данных, что также отмечается в статье [3], где автор предлагает использовать файловую систему для хранения и быстрого восстановления резервных копий виртуальных машин. Также, помимо того, что ZFS поддерживает атомарные операции над файлами и создание снапшотов за константное время, в этой файловой системе реализовано сжатие данных, что позволит эффективнее использовать дисковое пространство [12].

Транзакционный доступ к файлам, используя файловую систему ZFS, будет организован следующим образом: каждая версия файла будет иметь свой снапшот. Каждый снапшот в ZFS является read-only, поэтому изменить снапшот и получить состояние гонки за одним ресурсом из разных потоков не получится. При необходимости получить данные файла или внести в него изменение, используется последний снапшот, ассоциированный с нужным файлом. А после внесения нужных изменений, создается новый снапшот.

4. Реализация

4.1. Архитектура библиотеки

Основная часть библиотеки состоит из нескольких модулей:

- ZFSLibrary: Это основной компонент библиотеки, который обеспечивает взаимодействие с файловой системой ZFS и системой версионирования. Он служит в качестве посредника между файловой системой и системой версионирования, обеспечивая координацию и управление всеми операциями, и является входной точкой взаимодействия с библиотекой.

- ZFSHandler: Этот компонент представляет собой обработчик для файловой системы ZFS, которая используется для хранения и управления файлами и снапшотами. Он обеспечивает функции для сохранения файлов и создания снапшотов, а также для взаимодействия с ними. Помимо этого, в этом модуле реализован механизм итерации по файлам хранилища, который возвращает абстрактный итератор для каждого файла, с помощью которого можно получить мета-данные файла, такие как название, размер, дату создания и прочие. Предполагается, что данная библиотека должна работать с том числе с файлами больших размеров, поэтому для получения содержимого файла этот модуль возвращает его дескриптор, по которому итеративно можно получать содержимое файла и сохранять в нужное место на клиентской стороне.

- VersionManager: Этот компонент отвечает за назначение версий и управления ими посредством снапшотов. В этом модуле присутствуют такие методы, как createVersion() и deleteFileVersion(), которые используются для создания и удаления версий файлов. Каждая версия представляет собой уникальное состояние файла и содержит его снимок в определенный момент времени.

- File: Эта абстракция представляет собой файл, который сохраняется в файловой системе ZFS. Он содержит информацию о назва-

нии файла и пути до него в файловой системе ZFS.

- Snapshot: Этот компонент представляет собой снимок файла, созданные в файловой системе ZFS. Он содержит название снапшота в файловой системе, с помощью которого можно восстановить состояние файла на момент снимка.

- VersionedFile: Этот компонент представляет собой версию файла, который отслеживается системой версионирования. Он содержит информацию о версии файла и ссылку на соответствующий снапшот.

- ErrorHandler: Этот класс управляет всеми ошибками, которые могут возникнуть, и предоставляет информативные сообщения об ошибках.

- Logger: Этот класс отслеживает и записывать все действия, выполняемые в библиотеке, что может быть полезно для отладки и аудита.

Диаграмма классов описанного решения представлена на Рис.3.

4.2. Детали реализации

В качестве языка программирования для реализации был выбран язык C++, так как библиотека должна будет в дальнейшем использоваться для написания расширения для PostgreSQL, а расширения для этой СУБД пишутся в основном на языке C. Также в библиотеке предусмотрена прослойка, позволяющая использовать её из языка C. Для взаимодействия с файловой системой ZFS используется библиотека OpenZFS[7].

Класс ZFSHandler инкапсулирует взаимодействие с файловой системой ZFS. Этот модуль позволяет записать бинарный файл в файловой системе и сделать снапшот. Когда вызывается функция изменения

[7] `https://github.com/openzfs/zfs`

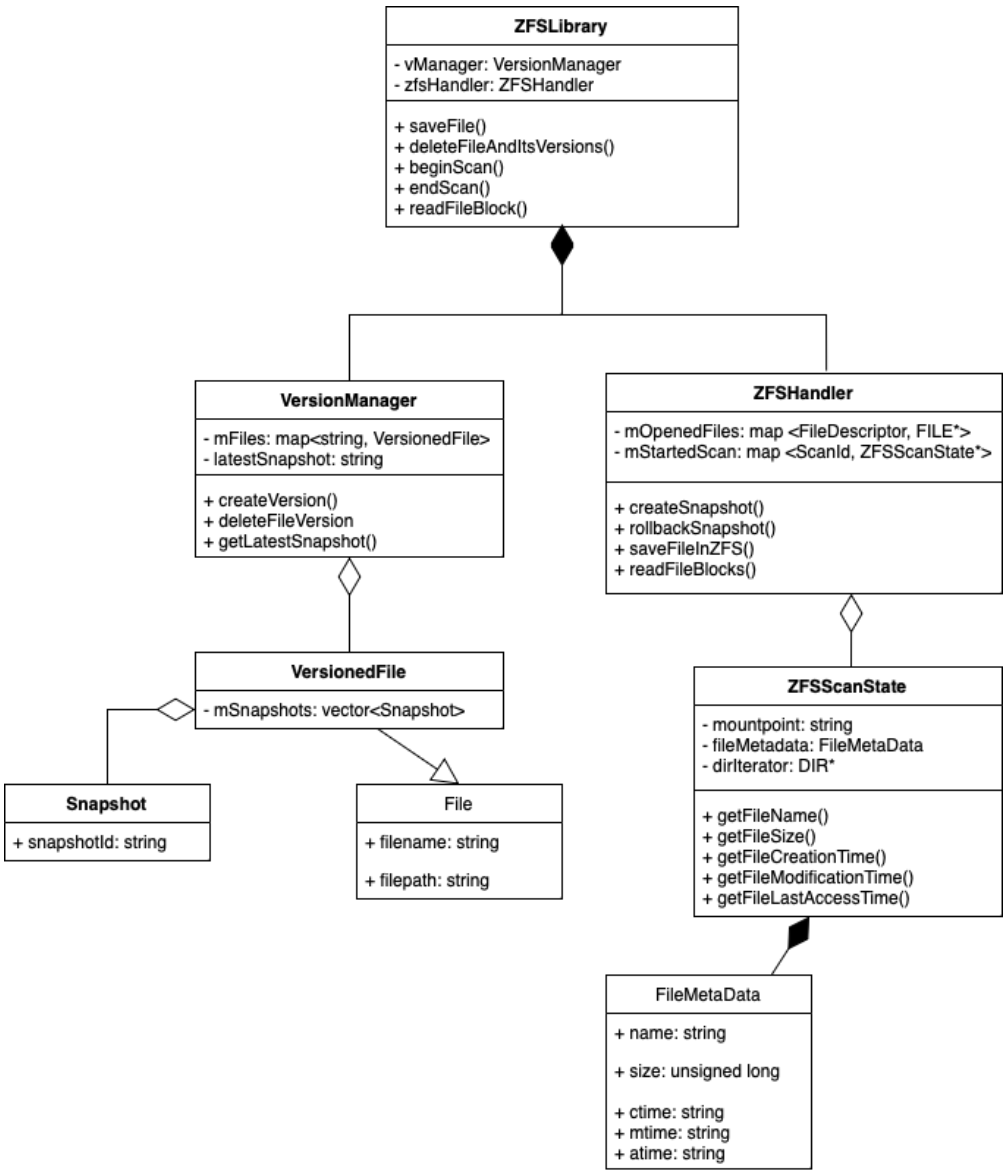

Рис. 3: Диаграмма классов UML

файла, то после применения изменений генерируется новый снапшот. В случае, когда необходимо восстановить какую-либо версию файла, этот модуль восстанавливает состояние файловой системы по снапшоту и возвращает содержимое файла. Также в модуле реализованы функции по удалению снапшотов и удалению файлов.

4.3. Разграничение адресного пространства

Разрабатываемая библиотека будет использоваться для написания расширения для СУБД PostgreSQL, которая работает в пользователь-

ском адресном пространстве. В этом пространстве программы ограничены в доступе к системным ресурсам и аппаратному обеспечению. Соответственно, библиотека должна выполнять все свои функции без необходимости обращаться к системным вызовам ядра операционной системы. Однако выполнение некоторых основных функций файловой системы ZFS, которая интегрирована в ядро операционной системы, происходит на уровне ядра. Операцией, которая требует выполнения на уровне ядра, является, например, создание снапшота.

Чтобы решить эту проблему, было решено реализовать ещё одну программу, которая будет запущена из-под суперпользователя и выполнять узкий перечень команд файловой системы ZFS, которые требуют выполнения на уровне ядра, например, создание нового снапшота или монтирование существующего снапшота. Реализовано это при помощи межпроцессного взаимодействия, обеспечивая быструю передачу сообщений, следующим образом: программа, запущенная из под суперпользователя, слушает очередь сообщений POSIX[8], в которую основной процесс библиотеки пишет заранее определенные команды, которые могут содержать параметры, разделенные пробелами. При появлении сообщения в очереди, программа читает и выполняет соответствующую операцию файловой системы ZFS, затем результат выполнения пишется в другую очередь, предназначенную для ответов на команды. Таким образом, процесс библиотеки, запущенный из-под непривилегированного пользователя, может выполнять операции, требующие выполнения в ядре операционной системы, и у пользователя СУБД появляется возможность выполнять все необходимые операции с файловой системой ZFS, оставаясь в пользовательском адресном пространстве. Диаграмма последовательности взаимодействия при создании нового снапшота ZFS продемонстрирована на Рис.4. На диаграмме зеленым цветом отображен экземпляр библиотеки, запущенный без прав суперпользователя, а оранжевым — экземпляр программы, запущенный из-под суперпользователя и выполняющий команду создания снапшота, обращаясь к файловой системе ZFS при помощи библиотеки OpenZFS. После

[8] `https://www.man7.org/linux/man-pages/man7/mq_overview.7.html`

выполнения команды, ZFSRootServer возвращает логическое значение, отображающее успех выполнения операции.

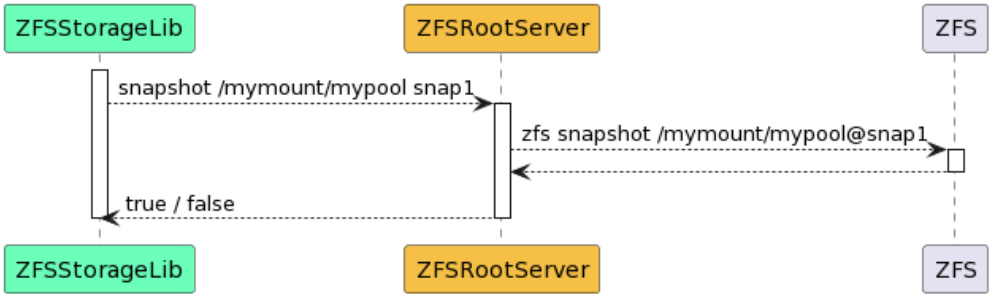

Рис. 4: Диаграмма последовательности UML создания нового снапшота

5. План по тестированию

5.1. Тестирование функциональности

Для тестирования функциональности библиотеки, а именно удовлетворения требованиям ACID, будет разработана отдельная программа, которая использует разработанную библиотеку и выполняет различные операции с файловым хранилищем в нескольких потоках, тем самым эмулируя реальные сценарии использования. Во время выполнения тестов будет проверяться, что при отмене транзакции файловое хранилище останется в нетронутом состоянии, а при фиксации транзакции все изменения будут применены к хранилищу. Помимо атомарности транзакций будет проверяться согласованность данных в файле и изолированность каждой транзакции, то есть тот факт, что новые транзакции не видят незафиксированные изменения других транзакций. Таким образом будут составлены и автоматизированы основные тест-кейсы, позволяющие проверить функционирование библиотеки должны образом при многопоточном её использовании.

5.2. Интеграционное тестирование

Будет проведено тестирование производительности расширения, а именно тест скорости работы с файлами в СУБД: будут проведены эксперименты по сохранению, изменению и извлечению файлов из базы данных, используя встроенные механизмы PostgreSQL (bytea, BLOB), а также с использованием разработанного расширения, и будут замерены такие метрики, как время выполнения запроса и количество транзакций в секунду (TPS).

Заключение

В результате работы над производственной практикой были решены следующие задачи:

- Изучены существующие подходы к решению задачи хранения бинарных данных в PostgreSQL.
- Определён подход к хранению бинарных данных.
- Реализована первая версия библиотеки для написания расширения PostgreSQL.

Планы для дальнейшей работы:

- Провести тестирование полученного решения.

Код проекта закрыт и принадлежит компании ООО “Датаджайл”.

Список литературы

[1] AcidFS documentation // Pylons project. — Access mode: https://docs.pylonsproject.org/projects/acidfs/en/latest/ (online; accessed: 21.01.2024).

[2] Albe Laurenz. Binary data performance in PostgreSQL // Cybertec, professional partner for PostgreSQL services, support and training. — 2020. — Access mode: https://www.cybertec-postgresql.com/en/binary-data-performance-in-postgresql/ (online; accessed: 24.05.2023).

[3] Angkaprasert Tinnaphob and Chanchio Kasidit. A Backup Mechanism of Virtual Machine Checkpoint Image using ZFS Snapshots // 2023 20th International Joint Conference on Computer Science and Software Engineering (JCSSE). — 2023. — P. 506–511.

[4] BinaryFilesInDB // PostgreSQL Wiki. — 2021. — Access mode: https://wiki.postgresql.org/wiki/BinaryFilesInDB (online; accessed: 25.11.2022).

[5] Armbrust Michael, Das Tathagata, Sun Liwen, Yavuz Burak, Zhu Shixiong, Murthy Mukul, Torres Joseph, van Hovell Herman, Ionescu Adrian, Łuszczak Alicja, undefinedwitakowski Michał, Szafrański Michał, Li Xiao, Ueshin Takuya, Mokhtar Mostafa, Boncz Peter, Ghodsi Ali, Paranjpye Sameer, Senster Pieter, Xin Reynold, and Zaharia Matei. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores // Proc. VLDB Endow. — 2020. — aug. — Vol. 13, no. 12. — P. 3411–3424. — Access mode: https://doi.org/10.14778/3415478.3415560 (online; accessed: 11.01.2024).

[6] Wright Charles, Spillane Richard, Sivathanu Gopalan, and Zadok Erez. Extending ACID semantics to the file system // TOS. — 2007. — 06. — Vol. 3.

[7] Fittl Lukas. 5mins of Postgres E3: Postgres performance cliffs with large JSONB values and TOAST // pgAnalyze Blog. — 2022. — Access mode: https://pganalyze.com/blog/5mins-postgres-jsonb-toast (online; accessed: 02.01.2023).

[8] Hilliger von Thile Alexander Melzer Ingo. Smart files: combining the advantages of DBMS and WfMS with the simplicity and flexibility of spreadsheets // Gesellschaft für Informatik e.V. — 2005. — Access mode: https://dl.gi.de/items/0a610b84-20ad-4366-9a5b-e9c1e03f2ce9 (online; accessed: 10.01.2024).

[9] Paolucci Nicola. How to handle big repositories with Git // Atlassian tutorial. — Access mode: https://www.atlassian.com/ru/git/tutorials/big-repositories (online; accessed: 07.01.2024).

[10] TOAST // PostgreSQL Documentation. — 2015. — Access mode: https://www.postgresql.org/docs/current/storage-toast.html (online; accessed: 25.11.2022).

[11] Vondra Tomas. Postgres vs. File Systems: A Performance Comparison // EDB Blog. — 2022. — Access mode: https://www.enterprisedb.com/blog/postgres-vs-file-systems-performance-comparison (online; accessed: 02.01.2023).

[12] Меликов Георгий. ZFS: архитектура, особенности и отличия от других файловых систем // «Завтра облачно», журнал о цифровой трансформации от VK Cloud Solutions. — 2020. — Access mode: https://mcs.mail.ru/blog/zfs-arhitektura-osobennosti-i-otlichija (online; accessed: 02.01.2023).