Федеральное государственное автономное образовательное учреждение
высшего образования «Национальный исследовательский университет
«Высшая школа экономики»

Факультет компьютерных наук
Основная образовательная программа
Прикладная математика и информатика

КУРСОВАЯ РАБОТА

Программный проект на тему

"Разработка технологии транзакционного управления данными на основе репликации и сетевого консенсуса в вычислительном облаке"

Выполнил студент группы 175, 3 курса,
Никаноров Кирилл Владимирович

Руководитель КР:
Кандидат технических наук
Руководитель подразделения в Яндекс.Облако
Бородин Андрей Михайлович

Куратор:
Кандидат технических наук
Доцент
Сухорослов Олег Викторович

Москва 2020

Содержание

Аннотация

Реляционные базы данных высоко востребованы и применяются во многих развитых отраслях экономики всего мира. Огромное количество людей и приложений каждый день взаимодействуют с базами данных, поэтому важно использовать общий формат. Для того, чтобы можно было исследовать сложные зависимости между данными, очень хорошо подходит реляционная модель. В ситуации, когда важно сохранить все записи, а не только их часть, основные требования, предъявляемые к СУБД - транзакционность и ACID-compliance (1). Многие приложения высоко чувствительны к доступности СУБД, поэтому, помимо описанных требований, зачастую необходимо реплецировать данные в разные зоны доступности, чтобы повысить up-time сервиса. В данной работе пойдёт речь преимущественно именно о таких СУБД. Результат работы - приложение (база данных), соответствующее описанным требованиям и предоставляющее описанные гарантии.

Annotation

Relational databases are highly demanded and applied in all the developed economic sectors of the world. Huge amount of people and applications interact with databases every day. Thus it is quite important to use common format. In order to research complex dependencies in data, relational model is convenient. Further we will consider situations, in which it is important to save all the data. Main requirements for DBMS - transactionality and ACID-compliance. Lots of applications highly depend on DBMS availability. Thus database replication often considered. It helps to increase service up-time via placing copies in different availability zones. Further we will consider that DBMS. Result of this coursework project is application - database, which provides describes requirements and guarantees.

Ключевые слова: реляционные, транзакции, ACID, доступность, СУБД, репликация, up-time

1 Введение

Перед тем, как приступить к основной части работы, приведу необходимые для понимания определения:

Реляционная модель данных - это модель данных, в которой все данные представлены в виде кортежей, кортежи сгруппированы в relations (можно перевести как "отношения"). Можно думать, что relations - таблица в базе данных, кортеж - список атрибутов объекта, строка в таблице. Для данной работы такого понимания вполне достаточно.

Транзакция в программировании - набор изменений в базе данных (логическая единица), которые либо будут применены полностью, либо вовсе не будут применены к системе. Результат не зависит от параллельно выполняющихся транзакций. Таким образом, в случае отказа оборудования, система всегда будет находится в некотором согласованном допустимом состоянии.

Доступность БД - возможность работы с базой данных в случае временного разделения сети.

СУБД (система управления базами данных) - комплекс программ, обеспечивающий возможность создания и работы с базой данных

Репликация - копирование данных из одного источника в другой. Здесь и далее подразумевается репликация базы данных, изменения реплицируются без какой-либо задержки. Primary node - основной узел с БД, standby - реплика

Up-time сервиса - показатель, отражающий, какой процент всего времени (как правило, за месяц) сервис запущен и обрабатывает запросы

Узел сети - Обозначение произвольного компьютера с подключенной компьютерной сетью. Сеть может как и частично, так и вовсе перестать быть доступна для некоторого подмножества участников сети в произвольный момент.

1.1 Постановка задачи

Задача, решаемая этим курсовым проектом - разработка SQL реляционной транзакционной отказоустойчивой БД, ACID-compliant. Предполагается сценарий, когда несколько узлов в сети занимаются работой с базой данных. Попробуем получить лучшую производительность, чем с обычной локальной записью изменений - как это делается в большинстве баз данных. Отказоустойчивость в данном контексте - скорейшее продолжение корректной работы в случае отказа нескольких узлов сети. Причиной отказа могут быть как и проблемы с сетью, так и проблемы аппаратного уровня. Более точное описание ограничений будет позднее. Отказоустойчивость будет достигаться во многом с помощью репликации. При том важно сохранить транзакционность и ACID-compliance. Понятно, что писать СУБД с нуля очень затратно, поэтому я буду рассматривать возможность модификации кода существующей СУБД.

1.2 Актуальность и значимость

Любое современное приложение использует базы данных. Это может быть как сбор статистики (хотя для этой задачи иногда лучше подходят другие базы данных, предоставляющие BASE (2) гарантии), так и данные для авторизации пользователей, и т.д. Выше перечисленное является необходимым для абсолютного большинства приложений. Также для большей корректности и удобства в программировании транзакционные БД много предпочтительнее в использовании. Зачастую данные необходимо реплицировать - это могут быть просто ценные данные, потеря которых может стоить очень дорого (денежные счета, облачное хранилище важного клиента). Также репликация помогает увеличить географическую доступность, уменьшить время доступа.

2 Обзор существующих решений

В этом разделе я затрону как и ACID-решения, так и другие. Стоит понимать, что выбор СУБД во многом зависит от задачи, так и от возможностей (возможность запустить базу на собственных вычислительных ресурсах, либо воспользоваться платными решениями и разместить базу данных в облаке). В любом случае, рассматривать решения будем с точки зрения сформулированной ранее задачи.

2.1 Open source БД

2.1.1 PostgreSQL

PostgreSQL(4) - Реляционная транзакционная СУБД, ACID-compliant. Есть возможность настройки как и синхронной, так и асинхронной репликации. Рассмотрим асинхронную репликацию более подробно: Из плюсов, на Primary Node нет дополнительной задержки, так как не нужно ждать подтверждение, что транзакция записалась на standby. С другой стороны, в случае отказа Primary Node после подтверждения транзакции, но до её отправления на standby, клиенту уже пришло подтверждение транзакции, но на самом деле данные были утеряны в случае отказа одного узла сети. Таким образом, часть транзакций может быть утеряна. Это нехорошо сказывается на отказоустойчивости. С другой стороны, синхронная репликация не имеет такого недостатка. Она, в свою очередь, оказывает большое влияние на общую производительность системы в негативную сторону. С точки зрения рассматриваемой задачи синхронная репликация подходит больше. Но в случае большего числа реплик всё становится очень медленным, поэтому такой вариант не подходит. Более того, локальная запись на диск - является самым узким местом, а одно из задач данной работы - улучшить производительность в сравнении с обычной записью на диск, поэтому этот вариант не совсем нам подходит.

2.1.2 MySQL

MySQL(3) - Реляционная СУБД, ACID-compliant. Ситуация в целом такая же, как и в PostgreSQL.

2.1.3 ClickHouse

ClickHouse(5) - Колоночная СУБД, предназначена для аналитической работы с данными. Язык представляет собой некоторое подмножество SQL, например, запрещены удаления. Система решает совсем другую задачу, поэтому нет смысла в дальнейшем её рассмотрении.

2.1.4 MongoDB

MongoDB(6) - документо-ориентированная СУБД, не ACID-compliant, noSQL. Данные представлены в похожем на json формате.
Присутствует Community edition с открытым исходным кодом. Аналогично, нет смысла в дальнейшем рассмотрении. С другой стороны, интересен тот факт, что без большой дополнительной настройки СУБД масштабируется горизонтально. Впрочем, отказоустойчивости это не добавляет, а вот доступности - вполне.

2.1.5 Apache Cassandra

Apache Cassandra(7) - колоночная распределенная СУБД, предоставляет BASE гарантии, noSQL (свой язык запросов CQL - Cassandra query language). Очень хорошо масштабируется. Из вышеописанного следует, что эта система также нам не подходит.

2.2 Платные решения

2.2.1 Oracle

Oracle Database(8) - облачный сервис, много-модельная (в том числе реляционная) СУБД. SQL, ACID-compliant. Поскольку Oracle не раскрывает детали реализации, довольно сложно что-либо заключить об отказоустойчивости системы.

2.2.2 DynamoDB

Amazon DynamoDB(9) - облачный сервис, документоориентированная, а также key-value СУБД. Предоставляется в рамках AWS - amazon web services. Соответственно, noSQL язык запросов, не ACID. Точнее, предоставляет только 'C' и 'D' из ACID. Отказоустойчивость и высокая доступность достигается за счёт синхронной репликации между разными датацентрами. Есть публикация (10), в которой подробно описывается устройство БД.

2.2.3 Amazon Aurora

Amazon Aurora(11) - облачный сервис, реляционная транзакционная СУБД. SQL, ACID. Высокая доступность и отказоустойчивость достигается за счёт репликации. Вместо чтения с локального хранилища Primary node выполняет чтение с реплик. За счёт этого обеспечивается лучшая производительность, чем при локальном чтении. Подробнее про это далее. Репликация на основе сетевого консенсуса, на выходе получаются гарантии близкие к синхронной репликации, но производительность системы выше, чем при синхронной репликации. Есть публикация (12), в которой подробно описывается устройство СУБД. Если бы это решение было бесплатным, то оно отлично решало бы поставленную задачу.

2.2.4 Microsoft Azure

Microsoft Azure(13) - облачный сервис для работы с БД. Ситуация в целом аналогична Amazon Aurora. Более того, в публикации (14) Microsoft реализуют идею Amazon Aurora. Это показывает, что такое вполне возможно написать за относительно небольшой срок. Аналогично, решение прекрасно решало бы поставленную задачу, если бы оно было бесплатным.

2.3 Результаты анализа

Как видно, на рынке представлено очень много различных баз данных. Это говорит о том, что СУБД высоко востребованы. Практически для каждой задачи можно найти подходящую БД. Среди решений нет подходящего для нашей задачи, но есть Amazon Aurora - платное решение, но решающее поставленную задачу. Основные идеи представлены в статье. Хоть и исходный код авторы не предоставляют. Далее я буду реализовывать идеи Amazon Aurora, чтобы выполнить поставленную задачу.

3 Разбор статьи

3.1 Обоснование выбора

Как было сказано в выводах анализа, статья (12) полностью освещает идеи, необходимые для реализации СУБД, решающей поставленную задачу. Анализ существующих решений не только показал, что такой СУБД нет, но и показал, где можно искать вдохновение для разработки. Поэтому я буду воспроизводить Amazon Aurora по публикации (12). Для начала подробно разберём основные идеи статьи. Параграф 3 полностью посвящается анализу статьи.

3.2 Определения

Здесь будут приведены некоторые важные определения

WAL - write ahead logging, журнал изменений в базе данных, генерируется после каждого запроса к базе, подробнее в следующем разделе.

Primary node - Узел сети, на котором выполняются все вычисления. Клиенты подключается именно к этому узлу. Primary node не хранит состояния, про его устройство в соответствующем разделе.

Storage node - Узел сети, реплика Primary Node. Непосредственно хранит базу данных.

AZ - Availability zone, зона доступности, изолированные друг от друга. Таким образом, очень мало вероятно, что некоторое глобальное событие произойдёт сразу в нескольких AZ. Например, отключение электричества, падение метеорита. Всё это достаточно локальные события и вряд ли смогут повлиять на разные AZ. *LSN* - Log sequence number, глобальный номер изменения из журнала WAL, всё время возрастает. С его помощью можно выбрать самую актуальную версию объекта.

3.3 Репликация БД по сети

Логичный вопрос, который возникает, как лучше всего организовать репликацию БД. Основные требования - минимизация нагрузки на сеть, а также применение изменений, внесенных в базу, без задержки на все реплики. То есть при каждом изменении в базе нужно сразу его сообщать всем репликам, при том в некотором сжатом формате. Авторы в статье рассматривают MySQL. Можно думать, что Amazon Aurora - надстройка над MySQL. в MySQL работа с диском - работа с блоками (страницами) фиксированного размера. Поэтому база на самом низком уровне представляет собой некоторую иерархию файлов, где каждый файл - целое число страниц некоторого размера (16Кб). Любое изменение в базе данных можно представить как изменение некоторого числа страниц. Будем при каждом запросе генерировать набор изменений, которые нужно применить к некоторому набору страниц. Получается, на реплицирующей стороне не нужно выполнять дополнительных вычислений, а просто применять изменения к файлам. И такой способ кодирования оказывается очень компактным. Такой журнал изменений называется WAL - Write ahead logging. Именно этим способ будем выполнять репликацию - применяя WAL на все standby. Есть ещё одна тонкость с checkpoint, про это будет далее. С помощью такого журнала можно представить всю базу данных. Более того, если реплика какое-то время не применяла WAL, восстановить БД очень просто. Присвоим каждому изменению некоторый номер, реплика будет присылать номер последнего примененного ею изменения, а Primary node в ответ будет стримить (потоковая передача) WAL с нужного момента.

3.4 Иерархия сети

БД представляет собой Primary node и 6 Storage node. Primary node не хранит никакого состояния, все данные хранятся на Storage node. Storage node не является полной репликой базы, но с помощью некоторого числа

Storage Node можно получить полную версию базы данных. 6 Storage node располагаются в 3-ёх AZ. Далее более подробно.

3.5 Алгоритм консенсуса

Опишем алгоритм консенсуса, который нам понадобится в дальнейшем. Пусть есть некоторый узел сети, который желает записывать и читать некоторые объекты в распределенную систему, состоящую из 6 узлов. Тогда действовать будем так: Осуществляем запись во все 6 узлов сети, но когда из 4 пришло подтверждение записи, считаем что всё записано. В случае чтения, читаем из 6 узлов, как только было прочитано 3 версии объекта, завершаем операцию. Этого достаточно, чтобы восстановить последнюю версию. Представим, что у объектов есть номер, по которому можно определить кто из них был записан самым последним. Тогда среди 3 объектов выберем самый последний. Утверждается, что это будет самая актуальная версия объекта. Действительно, известно, что хотя бы в 4 узлах записана последняя версия объекта, поэтому при чтении из 3-ёх узлов, обязательно хотя бы один узел содержит последнюю версию. Таким образом, при отказе двух узлов сети система сможет продолжить корректно работать, а при отказе 3-ёх - корректно обрабатывать чтение.

3.6 Primary node

Итак, клиентские соединения устанавливаются в Primary node. С точки зрения клиента это обычный SQL сервер, работающий на некотором endpoint в сети. Различия наступают в момент записи изменений на диск. Как до этого было сказано, Primary node не хранит состояния, все изменения записываются в 6 Storage node. Применяется выше описанный алгоритм консенсуса, то есть для каждого изменения WAL запись дожидается только 4 подтверждений. В свою очередь, когда на Primary node вызывается операция чтения страницы с диска - вместо этого выполним чтение 3-ёх версий страницы со

Storage node и выберем последнюю (с помощью LSN). Таким образом, Primary node представляет собой обычный MySQL с подмененными операциями работы с диском, применяющий немного измененную репликацию на 6 Storage node.

3.7 Storage node

Storage node - обычный MySQL, работающий в режиме репликации, при том на нём ещё выполняется сервис, который читает запрашиваемые страницы и отправляет их по сети (на Primary node).

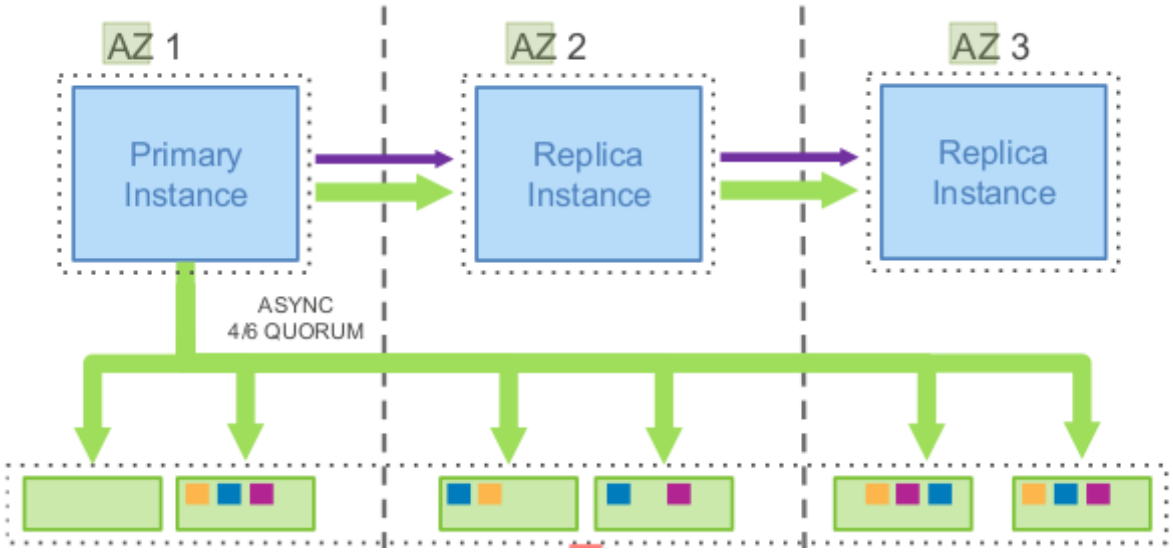

Рис. 3.1: Архитектура сети Amazon Aurora, стрелочки - передача WAL

3.8 Устройство всей системы

Итак, обобщим всё выше сказанное. Все узлы сети представляют собой изменённый MySQL. Выше приведена поясняющая схема. Пока что не будем вдаваться, что такое Replica instance, об этом будет позднее.
Опишу последовательность действий:

- Клиент подключается к Primary Instance (node)
- Клиент формирует запрос к базе данных и отправляет его на Primary node

- В случае записи:
  - Primary node генерирует WAL, соответствующий запросу клиента
  - Primary асинхронно рассылает поочередно все изменения WAL всем Storage node, для каждого дожидаясь только 4 подтверждений.
  - Storage node, получившие изменения из WAL, применяют его, изменяя (создавая, удаляя) соответствующие страницы
  - Storage node, получившие изменения из WAL и применившие его, отправляют подтверждение
- В случае чтения:
  - MySQL вызывают функцию для чтения страницы с диска на Primary node (возможно несколько раз и из разных процессов)
  - Primary асинхронно рассылает запрос на чтение страницы всем Storage node, дожидаясь 3-ёх ответов
  - Storage node, получив запрос на страницу, читает её с диска и отправляет по сети в Primary node
  - Primary выбирает страницу с самым большим LSN и возвращает её функции, вызвавшей чтение этой страницы

3.9 Мотивация

До этого было лишь описание системы, но не было объяснений почему именно такая архитектура. Попробую раскрыть мотивацию. Число 6 выбрано почти случайно, важно чтобы было достаточно много узлов для быстрой работы, а также чётное число узлов, чтобы выполнять чтение можно было с половины узлов. Primary node не хранит состояния из соображений отказоустойчивости. Если откажет не более двух Storage node, система без проблем продолжит корректную работу. В случае же отказа Primary node её можно просто перезапустить на другом узле сети. Поэтому down-time СУБД почти

полностью отсутствует. Консенсус, очевидно, быстрее синхронной репликации, но есть ли здесь прирост в сравнении с записью на диск? Рассуждение примерно такого характера: "В данной системе запись на диск просто переносится в другое место, но ещё добавляется задержка от сети, поэтому всё будет работать медленнее". Это не так, потому что:

- Storage node не обязательно выполнять запись на диск, они могут сохранить страницу у себя в кэше и потом её записать на диск.
- Чтение страниц можно выполнять параллельно: например, каждый раз выбирать 3 разные Storage node. Тогда операции с накопителем распределяются и уже нет упора в пропускную способность одного накопителя, а у сети намного большая пропускная способность, чем у записи на диск.

Результирующая система сохраняет все преимущества MySQL, при том ещё добавляется отказоустойчивость и доступность (про доступность далее). Действительно, при операции чтения/записи с диска могла итак произойти ошибка, поэтому из-за работы с сетью новых мест для возможной ошибки не добавилось. Транзакционность обеспечивается за счёт более высокоуровневой логики, а хранилище разнесено по узлам сети.

3.10 Read only node

Для того, чтобы уменьшить нагрузку с Primary node, вводятся так называемые Read only node. Это узлы сети, которые принимают клиентские соединения и работают с запросами только на чтение. Их вводятся два. Запросы только на чтение - это очень частый сценарий использования, поэтому такой приём позволит существенно снизить нагрузку с Primary node. Read only node, также как и Primary node, устанавливают соединения со всеми Storage node и выполняют только асинхронное чтение 3/6. Здесь возникает небольшая техническая проблема. В MySQL для страниц ещё существует

локальный кэш, поэтому при записи в Primary node важно инвалидировать соответствующие страницы и на Read only node, чтобы везде была актуальная версия. Иначе Read only node не выполнит чтение со Storage node, потому что есть локальная версия страницы. Поэтому WAL нужно стримить (потоковая передача) в том числе и на Read only node, чтобы заменить страницы на актуальные. Передачу WAL на Read only instance можно видеть на схеме 3.1.

3.11 Конец параграфа

Здесь были описаны основные идеи Amazon Aurora, в данной работе я постараюсь применить самые ключевые из них

4 Выбор платформы

Как написано раннее, писать СУБД с нуля - довольно бессмысленная затея, поэтому я буду модифицировать некую базу данных с открытым исходным кодом. Из анализа существующих решений - подходят два варианта: PostgreSQL и MySQL. Я решил остановиться на PostgreSQL, по целому ряду причин:

- WAL репликация работает почти без дополнительной настройки
- Работа с диском осуществляется в терминах страниц
- Несмотря на то, что PostgreSQL написан на Си, в нём очень читаемый исходный код
- Существует большое количество модификаций, большое сообщество разработчиков, у которых можно изучить некоторые полезные идеи
- Присутствует очень качественная документация
- PostgreSQL - самая технологичная база данных

- Как будет понятно далее, изменения очень удачно накладываются на обычный PostgreSQL

5 Устройство PostgreSQL

OId - object identifier, 4-байтный идентификатор объекта

В этом параграфе будет обзор той части, PostgreSQL, которая необходима для понимания данной работы. Затронуты будут в основном низкоуровневые вещи, в частности, нас слабо интересует как устроена обработка запросов, но нас интересует, как осуществляется взаимодействие с накопителем. Большая часть информации о работе PostgreSQL взята отсюда (15)

5.1 БД, relations, Oid

Одна СУБД работает с несколькими базами данных. Клиент осуществляет подключение к конкретной базе данных. В каждой базе данных есть relations - таблицы. На диске relation представлен как несколько файлов, размер каждого из которых кратен 8 Кб - размер страницы, с которой работает PostgreSQL. У БД и у таблиц есть свой уникальный идентификатор - Oid

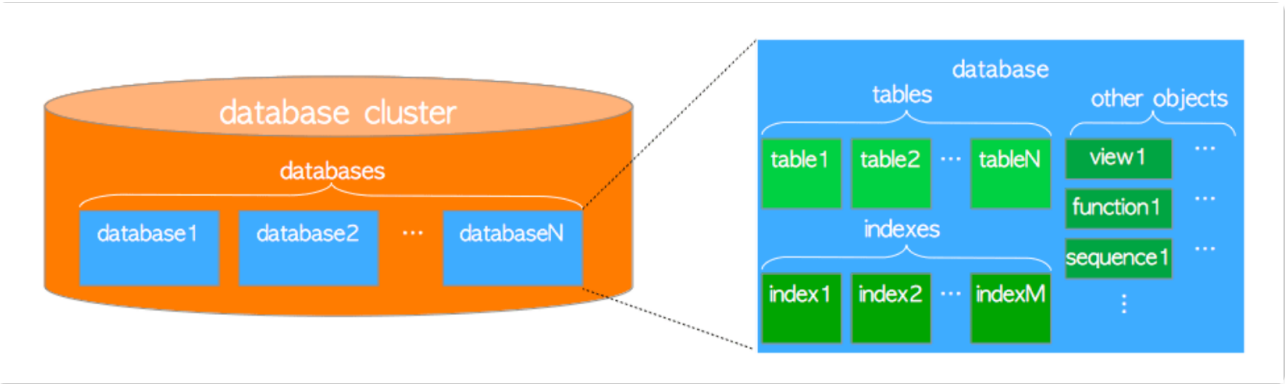

Рис. 5.1: Логическая структура кластера БД

5.2 Иерархия файлов

В директории с файлами базы данных - есть поддиректория base, в ней хранятся все файлы баз данных. в base находятся директории, в названии

которых соответствующий OId базы данных. Таким образом, путь к базе данных - PGDBPATH/base/DBOid. Внутри базы данных, помимо служебных файлов, лежат файлы relations. rOId - relation OId. Одному relation соответствует несколько файлов:

- rOId, rOId.1, rOId.2, ...- файлы, в которых хранится непосредственно таблица, на каждый новый занимаемый гигабайт создается новый файл.
- rOId_fsm - free space map - вспомогательный файл, ускоряет некоторые внутренние процессы
- rOId_vm - visibility map - вспомогательный файл, ускоряет внутренние процессы

Таким образом страница однозначно идентифицируется следующими числами: DBOid, rOId, BlockNumber - индекс страницы в файле, ForkNum - указывает на обычный файл, _fsm или _vm соответственно. С помощью такого небольшого числа байт можно сообщить информацию о желаемой к прочтению странице памяти.

5.3 Buffer manager

Buffer manager - непосредственно кэш страниц. При записи страница помещается в этот кэш и помечается как dirty, затем фоновый процесс записывает такие страницы на диск. С другой стороны, чтение страниц также осуществляется через этот кэш. В нём предусмотрены механизмы конкурентного доступа. Вот что происходит при чтении страницы из кэша:

- Если страница найдена в кэше, сразу возвращаем её и выходим из функции
- Если страницы нет, значит нужно освободить место в кэше и прочитать её с диска.
- Самая менее используемая страница удаляется из кэша.

- Запрошенная страница помещается в кэш (чтение с накопителя).
- Возвращаем её и выходим из функции.

5.4 Storage manager

Storage manager - набор функций, с помощью которых непосредственно осуществляется взаимодействие с файловой системой. В частности, кэш страниц использует функции Storage manager, чтобы прочитать нужную страницу.

5.5 Параллельное исполнение

PostgreSQL написан на Си стандарта 89 года и там нет потоков из соображений совместимости с огромным числом устройств. Вместо этого создаются разные процессы с помощью системного вызова fork(), а общая память выделяется через mmap().

5.6 Работа с сетью

Работа с сетью осуществляется только синхронными операциями, но это достаточно.

5.7 Применение WAL

Возникает вопрос, можно ли пропустить часть WAL и продолжить корректную работу (Если не будут запрашиваться пропущенные страницы). Ответ - да, так как изменения представляют собой перезапись части страницы, которая, как нам уже известно, однозначно идентифицируется. Поэтому можно вычислить её адрес, выделить место в файле, если его недостаточно и записать изменение в нужный файл. Аналогично, если файл отсутствует, его можно создать. Пропускать WAL необходимо для записи 4/6.

5.8 Процессы PostgreSQL

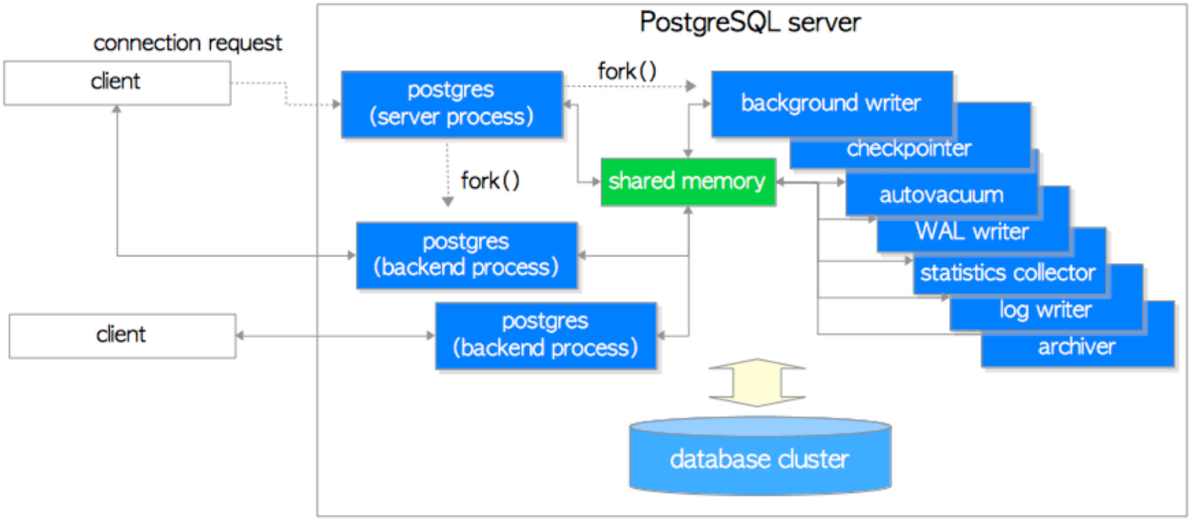

Рис. 5.2: Все запускаемые в PostgreSQL процессы

Самый главный процесс - postmaster. Он принимает новые соединения, создаёт backend для работы с данным клиентом (замечу, с одним клиентом может быть связано несколько процессов, если генерируемые им запросы достаточно объёмные). Получается, соединения с базой - отдельный процесс, достаточно дорогой объект. Далее, все процессы взаимодействуют через так называемую общую память, про неё было ранее. Опишем роль основных процессов:

- background writer - запись страниц, помеченных как dirty, на диск. dirty означает, что с данной страницей было произведена модификация, но изменения ещё не записаны на накопитель.

- checkpointer - процесс, создающий возможность восстановить состояние СУБД в некоторые моменты времени - в которых был создан checkpoint. Это достигается за счёт того, что после checkpoint каждая измененная страница первый записывается в WAL полностью, а не только набор изменений. За счёт этого, даже если произойдёт повреждение накопителя, БД можно будет восстановить полностью из WAL. При том не потребуется применять весь журнал, для каждой страницы можно начать с её последней полной записи.

- vacuum - процесс, который освобождает место, появившееся после удаления строки из таблицы. Таким образом, перемещая записи более компактно, иногда можно освободить несколько страниц.

- WAL writer - процесс, генерирующий WAL, а также же ответственный за его отправку репликах и сохранение на локальный диск.

6 Реализация с 1 Storage node

Далее будет описание моей реализации. В этом параграфе будет представлен список изменений для работы с 1 Storage node в синхронном режиме. Итак, чтобы начать работать в таком режиме нужно:

- Заменить операции работы с диском на удалённые вызовы

- При том нужно полностью удалить запись на диск - изменения применяются с помощью WAL, то есть оставить только отправку WAL по сети.

- Настроить PostgreSQL на Storage node для обработки удалённых вызовов

- Убрать лишние процессы

6.1 Замена операций диском

В PostgreSQL вся работа с диском происходит через некоторые оболочки над системными вызовами, поэтому можно просто заменить код функции на удалённый вызов, не изменяя точки вызова. Для работы с диском есть несколько видом запросов: Прочитать страницу, открыть файл, закрыть файл и т.д.
Их все нужно заменить на удалённые вызовы. Будем действовать так:

- Инициализируем socket для работы с сетью в первый вызов одной из таких функций
- Формируем структуру с необходимой информацией (тип запроса, идентификатор блока/файла)
- Отправляем запрос по сети на Storage node
- Читаем ответ и выходим из функции

6.2 Удаление записи

А операцию записи на диск необходимо убрать совсем, потому что записи уже применяются с помощью WAL, а локально Primary node не хранит состояния.

6.3 Настройка Storage node

На Storage node необходимо настроить несколько вещей:

- Синхронную репликацию
- Запустить сервис для работы с файловой системой через вызовы по сети.

Сервис будет работать примерно так:

- Создаём процесс на каждое новое соединение
- Читаем структуру по сети
- Выполняем запрос
- Отправляем ответ

Как видно, вся логика сосредоточена в Primary node, а Storage node синхронно обрабатывает поступающие запросы. Так будет и дальше.

6.4 Обобщение

Итак, все изменения идейно выглядят несложно, но реализовать их довольно нетривиально. Ещё большой вопрос, как правильно запустить сервис для работы с хранилищем по сети. По пути возникает большое множество технических проблем, но идейно все близко к описанному. Я привёл это описание, чтобы уже в случае 6 Storage node было много знакомых частей, более последовательное изложение материала.

7 Реализация с 6 Storage node

7.1 Storage node

Итак, в первую очередь заметим, что реализацию Storage node можно использовать из синхронного случая. Действительно, с точки зрения Storage node - это синхронные операции на чтение в одном процессе, на применение WAL в другом. Различие только в том, что часть WAL может быть пропущена. Но как было замечено, на корректность работы это не повлияет.

7.2 Logic node

В этой части будут большие изменения. Сначала обсудим чтение.

7.2.1 Чтение

Далее будет общее описание алгоритма чтения в случае работы с корутинами

- На каждый запрос страницы запускаем 6 корутин с 3-мя действиями:
  - Отправка запроса по сети
  - Чтение из socket запрошенной страницы - чтение ответа Storage node

  - Уведомление главного процесса о завершении операции

- После получения 3 уведомлений, главный процесс завершает остальные операции

- Выбирает самую актуальную страницу

- Возвращает её из функции

Существует множество библиотек, можно работать в терминах корутин. Корутины плохи тем, что в их случаи для каждого запроса создаётся отдельная корутина, которая будет дожидаться завершения операции. Это не очень удобно, хотелось бы пропускать некоторые операции чтения/записи, иначе получается нагрузка на сеть как в случае синхронной репликации. А отмена операций - тоже достаточно дорогая операция. Поэтому рассмотрим другие возможности.
Опишем алгоритм чтения в случае работы с потоками/процессами

- Запускаем 6 потоков, которые повторяют следующую последовательность действий

  - Читаем последний идентификатор страницы из памяти
  - Если эта страница уже была прочитана, засыпаем до уведомления
  - Иначе выполняем запрос, записываем ответ, уведомляем главный процесс

- В главном процессе Записываем идентификатор страницы в определенное место в память.

- Уведомляем спящие потоки, ожидающие нового запроса

- Собираем 3 уведомления и формируем ответ

В этом случае не только корректная работа, но и меньшая нагрузка на сеть. Представим ситуацию, когда доступ к 2-ум узлам сети пропал на небольшое

время. В этот период PostgreSQL мог запросить несколько страниц. Тогда два процесса, соответствующие этим узлам сети, даже не узнают об этих запрошенных страницах, потому что в это время они будут заблокированы на ожидание ответа от Storage node. Получается, нагрузка запросов на чтение равномерно распределяется между Storage node, учитывая пропускную способность сети.

Замечу, что в PostgreSQL можно добавить библиотеку pthread(17) и работать в терминах потоков, а не процессов. У этого подхода есть целый ряд минусов:

- Это технически непростая задача, использование новой библиотеки в таком большом проекте
- Внутренние функции PostgreSQL работают с процессами, поэтому надо будет очень аккуратно их использовать
- В ситуации, когда соединений устанавливается не так уж и много, можно и использовать отдельные процессы

Поэтому мной было принято решение работать с процессами, создавая процесс на каждое соединение. В конце параграфа будет представлен механизм уведомлений, упоминаемый в алгоритме.

7.2.2 Запись

Запись реализуем таким же алгоритмом, а точнее:

- Запускаем 6 потоков, которые повторяют следующую последовательность действий
  - Читаем последний сегмент WAL
  - Если этот сегмент уже был записан, засыпаем до уведомления
  - Иначе отправляем, дожидаемся ответа, уведомляем главный процесс

- В главном процессе Записываем очередной сегмент WAL в определенное место в память.
- Уведомляем спящие потоки, ожидающие нового сегмента
- Собираем 4 уведомления и подтверждаем, что запись сегмента завершена

Стоит понимать, что в случае записи важно либо записать изменения полностью, либо вообще не записать. Довольно опасно отправить одну часть сегмента, а затем другую. Поэтому такая модель отлично подходит для записи WAL: процессам отправляются цельные куски WAL'a, нагрузка также равномерно распределяется.

7.3 Механизм уведомлений

Выше был описан весь алгоритм в общих чертах. Из него хорошо видно, откуда выигрыш в производительности, почему сохраняется корректность и добавляется отказоустойчивость. Почему можно перезапустить Primary node без особых трудностей. В качестве механизма уведомлений нам понадобится несколько базовых примитивов синхронизации Semaphore(18) и Condition variable(19). В частности, с помощью семафора удобно отслеживать такие события как: наступление 3 первых событий из 6 - именно такого рода события возникает при асинхронном чтении/записи.
Condition variable, в свою очередь, позволяет реализовать механизм засыпания до наступления определенного события - в решаемой задаче запрос новой страницы или запись нового WAL.

8 CAP theorem

Логичный вопрос, который возникает, как предложенная система соотносится с CAP теоремой(20). PostgreSQL реализует от туда CAp - consistency

и availability. Замечу, что в описанной системе ситуация такая же. Действительно, в случае разделения сети на две равные части, запись становится невозможно сделать, при том Consistency и availability сохраняются.

9 Достигнутые результаты

Ссылка на репозиторий с кодом - в моей реализации реализована работа с одной Storage node, а также чтение 3/6.

10 Тестирование

Чтобы должным образом протестировать данную систему на производительность, необходим доступ к датацентру. У меня нет такой возможности, поэтому я только сделал тесты на корректность на своей локальной машине.

11 Возможные направления развития

Помимо всего, что реализовано, необходимо доделать запись 4/6, впрочем она во многом похожа на чтение, так что это не составит особого труда. Также осталось реализовать Read only node, у меня есть понимание, как они должны работать, код PostgreSQL на данный момент подготовлен на столько, что внесение такого рода изменений не составит труда. Также, стоит рассмотреть возможность создания новой Storage node из существующих - путём передачи того же WAL. Например, возникла такая ситуация, что одна Storage node отказала и необходимо сразу же начать создавать новую. Для этой задачи не хочется нагружать Primary node, поэтому нужно напрямую научиться взаимодействовать между Storage node. Помимо всего этого, необходимо добавить шифрование, потому что передачу между предполагаемыми AZ не является безопасным участком сети.

12 Заключение

В первую очередь, в мире не существует решения для сформулированной задачи с открытым исходном кодом, но есть платное решение и статья. С их помощью в этом курсовом проекте я попробовал реализовать описанную в статье идею, чтобы решить поставленную задачу. Сделать у меня это получилось не полностью, но я существенно продвинулся и имею полное понимание что нужно делать дальше. Была проведена колоссальная работа по анализу статьи, выбору платформы, изучению PostgreSQL, реализация алгоритма консенсуса, бесчисленное множество адаптаций PostgreSQL для поставленной задачи. Несмотря на это, реализовать идею из статьи Amazon Aurora вполне реально, но требуется много времени и усилий.

Список источников

1. Электронный ресурс Wikipedia, дата обращения 12.11.2019

2. Электронный ресурс acm, дата обращения 15.01.2020

3. Электронный ресурс MySQL, дата обращения 01.02.2020

4. Электронный ресурс PostgreSQL, дата обращения 04.11.2019

5. Электронный ресурс ClickHouse, дата обращения 01.02.2020

6. Электронный ресурс MongoDB, дата обращения 01.02.2020

7. Электронный ресурс Apache Cassandra, дата обращения 02.02.2020

8. Электронный ресурс Oracle Database, дата обращения 04.02.2020

9. Электронный ресурс Amazon DynamoDB, дата обращения 05.02.2020

10. DeCandia, Giuseppe & Hastorun, Deniz & Jampani, Madan & Kakulapati, Gunavardhan & Lakshman, Avinash & Pilchin, Alex & Sivasubramanian,

Swaminathan & Vosshall, Peter & Vogels, Werner. (2007). Dynamo: Amazon's highly available key-value store. Operating Systems Review - SIGOPS. 41. 205-220. 10.1145/1294261.1294281.

11. Электронный ресурс Amazon Aurora, дата обращения 11.11.2019

12. Verbitski, Alexandre & Bao, Xiaofeng & Gupta, Anurag & Saha, Debanjan & Brahmadesam, Murali & Gupta, Kamal & Mittal, Raman & Krishnamurthy, Sailesh & Maurice, Sandor & Kharatishvili, Tengiz. (2017). Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. 1041-1052.10.1145/3035918.3056101.

13. Antonopoulos, Panagiotis & Prakash, Naveen & Purohit, Vijendra & Qu, Hugh & Ravella, Chaitanya & Reisteter, Krystyna & Shrotri, Sheetal & Tang, Dixin & Wakade, Vikram & Budovski, Alex & Diaconu, Cristian & Saenz, Alejandro & Hu, Jack & Kodavalla, Hanuma & Kossmann, Donald & Lingam, Sandeep & Minhas, Umar. (2019). Socrates: The New SQL Server in the Cloud. 1743-1756. 10.1145/3299869.3314047.

14. Электронный ресурс Microsoft Azure, дата обращения 03.02.2020

15. Электронный ресурс interndb, PostgreSQL, дата обращения 11.11.2019

16. Электронный ресурс библиотека asio для c++, дата обращения 03.03.2020

17. Электронный ресурс библиотека pthread для языка Си, дата обращения 04.03.2020

18. Электронный ресурс Wikipedia, Semaphore. дата обращения 19.01.2020

19. Электронный ресурс Wikipedia, Condition variable. дата обращения 19.01.2020

20. Электронный ресурс Wikipedia, CAP theorem. дата обращения 04.11.2019