GeekBrains

Название курса

Урок 6. Оптимизация хранения данных: репликация, секционирование, сегментирование

На этом уроке:

- дадим определение OLTP и OLAP хранилищам данных;
- рассмотрим теоретические подходы к масштабированию систем хранения данных: вертикальное и горизонтальное масштабирование;
- изучим современные подходы и инструментарий для повышения масштабируемости, доступности и надежности систем хранения данных;
- выполним практическую работу: реализуем масштабируемую и надежную систему хранения данных социальной сети с повышенной доступностью.

Оглавление

Введение

В уроке “Анализ производительности, нагрузочное тестирование” мы определили, что производительность приложения ограничена производительностью подсистем, которыми оно пользуется. Вспомним, что Никлаус Вирт предложил рассматривать *программу*, как *структуры данных* плюс *алгоритмы* и распространить это определение на распределенный сервис. В данном случае, получаем, что *сервис* - это *система хранения данных* плюс *алгоритмы*. Следовательно, система хранения данных и будет нашим камнем преткновения, ограничивающим итоговую производительность сервиса.

В далеком 1983 году Джеймс Мартин в книге Managing the Data-base Environment определил четыре базовые операции для работы с данными, известные как операции *CRUD*:

- создание (create);
- чтение (read);
- обновление (update);
- удаление (delete).

Если оптимизировать скорость выполнения этих операций, можно существенно повысить производительность системы хранения данных. Оптимизация системы хранения данных - основная тема этого урока.

Теория урока

Системы хранения данных подразделяют на два класса:

- OLTP (On-line Transaction Processing) - выполняет обработку плотного потока простых операций изменения данных: создания, обновления и удаления - в режиме реального времени.
- OLAP (On-line Analytical Processing) - выполняет малое число операций создания или обновления, но использует сложные запросы чтения, включающие в себя функции агрегирования и анализа, на больших объемах данных, также известных как *BigData*

Важно понимать, что нет необходимости производить оптимизацию по всем операциям CRUD. Учитывая требования технического задания, нужно определить к какому классу относится ваша система хранения данных: OLTP или OLAP - и выбрать оптимизируемые в первую очередь операции.

Рассмотрим пример: пусть необходимо создать сервис для ведения бухгалтерской отчетности. Задача сервиса состоит в периодическом формировании сложных отчетов, использующих несколько таблиц базы данных (БД). В данном случае целесообразно использовать OLAP систему хранения данных, т.к. операция чтения должна быть оптимизирована по времени путем создания индексов под все пользовательские условия формирования отчетов. Создание записи будет сопровождаться обновлением всех индексов. Т.к. индексы представляют собой деревья, сложность данной операции в лучшем случае будет $O(log\ n)$, в худшем - $O(n)$.

Другой пример: система журналирования событий. События записываются в систему хранения данных с целью архивирования, т.е. запись должна осуществляться быстро плотным потоком. Здесь целесообразно использовать OLTP систему хранения данных. Легитимной оптимизацией будет являться полное удаление индексов. Сложность операций создания будет $O(1)$ - лишь системный вызов **`fsync.`**

Сложность выполняемых операций определена исходным техническим заданием. Оптимизация операций происходит за счет их ускорения, при известной сложности. Такую оптимизацию можно провести, ориентируясь не на функцию О-большое, а на функцию Тау - время выполнения. Дело в том, что сложности $O(n)$ и $O(\frac{n}{2})$ одинаковы, в то время как $T(n)$ в два раза выше чем $T(\frac{n}{2})$ и, разбив одно дерево индекса на два, сложность вставки данных так и останется величиной $O(n)$, но время

для вставки будет в два раза ниже - $T(\frac{n}{2})$. А для второго примера ускорения можно осуществить сменой диска с HDD на SSD. Как указано в работе Fsync Performance on Storage Devices выполнение операции **`fsync`** на SSD-диске занимает на порядок, а то и на два, меньше времени чем на HDD-диске: **0.14-9.6ms** против **17-66ms**. На этих примерах внимательный читатель заметит два подхода к выполнению оптимизации:

- **Вертикальное масштабирование** решает проблему ускорения системы хранения данных путем использование более быстрых подсистем: процессора с большей частотой выполнения операций, процессора с большим числом ядер, более быстрая оперативная память DDR4, использование большего объема оперативной памяти, замена HDD-диска на SSD-диск, подключение оптоволоконного сетевое соединение и т.п.

  Данный подход позволяет провести масштабирование быстро и, в большинстве случаев, дешево, но существует предел до которого можно им пользоваться. Например, мировой рекорд для частоты процессора был установлен процессором AMD FX-8370 в далеком 2014 и составляет 8.79 Ггц и если ваши сервера уже работают на нем, то вертикальное масштабирование - это не то решение, которое поможет вам в случае проблем с быстродействием процессора

- **Горизонтальное масштабирование** решает проблему ускорение системы хранения данных путем распределения нагрузки по нескольким подсистемам, работающим параллельно друг с другом: деление индекса между серверами при недостатке оперативной памяти на одном из них, разделение базы данных на несколько дисков при недостаточной скорости выборки данных с одного из них и т.п.

  Данных подход не быстрый, т.к. требует времени работы инженеров для реализации и поддержки; по этой же причине подход не будет дешевым. Но он позволяет масштабироваться независимо от границ технологического прогресса в области серверных компонент

Следует отметить, что вертикальное масштабирование является дешевым лишь до определенного предела. Согласно первому и второму законам Мура, экспоненциально растет не только производительность вычислительных систем, но и, косвенно, их стоимость. Самый дорогой 28-ядерный процессор Intel Xeon Platinum 8180 стоит больше $10K. За эти же деньги можно купить 15 процессоров AMD RYZEN 9 5900X, что даст 240 вычислительных ядер. По итоговой стоимости владения (Total Cost Ownership) видно, что в низком ценовом сегменте выгоднее использовать вертикальное масштабирование, а с переходом в верхний ценовой сегмент уместнее использовать уже горизонтальное масштабирование

При горизонтальном масштабировании с целью повышения доступности, надежности и масштабируемости систем хранения данных выделяют следующие подходы: репликация, секционирование и сегментирование. Рассмотрим их подробнее

Репликация

Репликация (replication) - это копирование информации на избыточные хранилища с целью повышения доступности и надежности системы хранения данных. Доступность достигается наличием возможности чтения разных копий данных из различных хранилищ. В случае выхода хранилища из строя, сохраняется работоспособность оставшихся хранилищ, с полной копией данных, способных принимать и обрабатывать запросы. Из этой особенности следует и надежность хранения данных, основной характеристикой которой является количество копий данных - *коэффициент репликации* (replication factor).

Поскольку речиь идет о копирований, хранилиза разделяют на два класса: *источник* копирования и *приемник*. *Источник* - хранилище первичного получения данных. *Приемник* - хранилище данных, скопированных из источника. В англоязычной литературе до недавнего времени источник было принято называть **master**, а приемник - **slave**. С 2013 года эта терминология подвергается попыткам пересмотра, а с 2020 года процесс начал лавинообразно ускоряться. В википедии добавлена отдельная статья по вариантам именования связки **master/slave**. Рекомендую ознакомиться для изучения современной англоязычной литературы. Далее будут использоваться всё-таки канонические термины.

С архитектурной точки зрения выделяют три подхода организации репликации:

- **master-slave** репликация подразумевает наличие одного источника в инфраструктуре. Остальные хранилища являются приемниками и не могут принимать запросы на создание, обновление и удаление данных
- **master-master** репликация подразумевает наличие нескольких равноправных источников в инфраструктуре. Суть репликации при этом не меняется: если запрос на создание, обновление или удаление данных попадает к любому из хранилищ, оно начинает временно выступать в роли источника и копирует данные на остальные хранилища, временно выступающие в роли приемника
- **Гибридное решение** предполагает совмещение master-master и master-slave репликаций совместо. В инфраструктуре присутствуют как равноправные источники, там и приемники доступные только для чтения

С алгоритмической точки зрения выделяют два типа репликации: *синхронная* и *асинхронная*. При синхронной репликации сервис источника дает положительный ответ на запрос изменения данных только после успешного изменения данных на всех приемниках (рисунок 1)

Рисунок 1

При асинхронной репликации источник дает положительный ответ на запрос изменения данных не дожидаясь завершения копирования данных на приемник (рисунок 2)

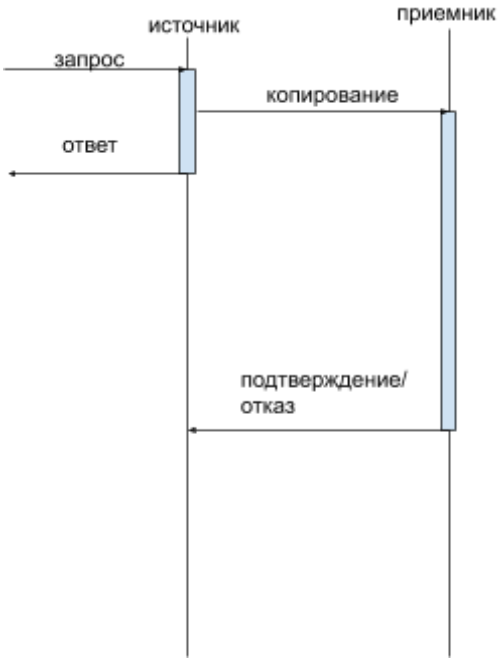

Рисунок 2

Синхронная репликация обладает определенной степенью надежности, однако при этом величина задержки выполнения операции определяется временем достижения нужного коэффициента репликации. Если репликация выполняется последовательно, то время выполнения запроса на изменение данных является суммой времен копирования данных на все приемники (формула 1). При

параллельной репликации, время определяется наибольшим временем копирования данных на приемник (формула 2)

$$t_{\text{операции}} = \sum_{i=0}^{n} t_i \quad (1)$$

$$t_{\text{операции}} = max(t_i),\ i = 0..n \quad (2)$$

Асинхронная репликация обеспечивает меньшие задержки на выполнении операций. Но она и не может гарантировать достижения нужного коэффициента репликации в случае отказа приемника

Секционирование

Секционирование (partitioning) - это разделение логически-независимых данных на секции для улучшения доступности и надежности. В качестве секции могут выступать таблицы, базы данных и хранилища на удаленных серверах. Доступность обеспечивается уменьшением значения Тау выполняемых операций - например, вставки в дерево индексов. В случае отказа одной из секций, остальные сохраняют свою дееспособность, что повышает надежность хранения данных.

При выполнении секционирования в первую очередь нужно определить, каким образом выделять секции в исходном хранилище данных. Современные РСУБД, такие как MySql, PostgreSQL, Oracle, поддерживают множество критериев секционирования данных. Основными являются следующие три:

- **По диапазонам значений** - секция определяется диапазоном допустимых значений. Напр., все записи с датами с 01 ноября 2020 года по 30 ноября 2020 года
- **По спискам значений** - секция хранения определяется конечным списком допустимых значений. Например, значения 1, 3, 5 сохраняется в первой секции; 2,4 - во второй
- **По хэш-функции** - секция хранения данных определяется возвращаемым значением заранее заданной хэш-функции

Исходная таблица

date	amount
2020.01.10	10.02
2020.01.15	12.2
2020.02.03	14.38
2020.02.20	16.56
2020.02.25	18.74
2020.03.04	20.92
2020.03.30	23.1
2020.04.05	25.28
2020.04.05	27.46
2020.05.20	29.64
2020.05.21	31.82

1

date	amount
2020.01.10	10.02
2020.01.15	12.2

date	amount
2020.02.03	14.38
2020.02.20	16.56
2020.02.25	18.74

...

2

date	amount
2020.01.10	10.02
2020.02.20	16.56
2020.03.30	23.1
2020.05.20	29.64

date	amount
2020.01.15	12.2
2020.02.03	14.38
2020.02.25	18.74
2020.03.04	20.92
2020.04.05	25.28
2020.04.05	27.46
2020.05.21	31.82

3

date	amount
2020.01.10	10.02
2020.02.20	16.56
2020.03.04	20.92
2020.03.30	23.1
2020.05.20	29.64

date	amount
2020.01.15	12.2
2020.02.03	14.38
2020.02.25	18.74
2020.04.05	25.28
2020.04.05	27.46
2020.05.21	31.82

Рисунок 3

> *Для самостоятельного изучения:* На рисунке 3 представлена исходная таблица платежей и варианты ее секционирования. Подскажите, по каким критериям выполнено секционирование в вариантах 1, 2 и 3? Проверьте правильность ответа на занятии, уточнив у преподавателя

Как видно на рисунке 3, разные критерии секционирования могут создавать секции разного размера относительно друг друга. Неудачный выбор критерия может привести к разбалансировке секций, как в примере номер 2, где первая секция почти в два раза меньше второй. Ситуация разбалансировки секций снижает позитивный эффект от применения данного подхода оптимизации системы хранения данных

Во вторую очередь нужно определить место хранения секций. Хранить секции можно в виде таблиц той же базы данных - это *вертикальное секционирование*. При подходе используется один сервер РСУБД, содержащий базу данных с несколькими секционированными таблицами. Позитивный эффект решения в этом случае обеспечивается разделением индексных деревьев, что ускоряет как чтение, так и вставку, обновление и удаление данных.

Кроме этого возможно появление эффекта естественного возникновения *активных* и *неактивных секций*. *Активная секция* - секция, данные которого необходимы приложению для выполнения операций в текущий момент времени. *Неактивная секция* - секция, хранящая архивные данные.

Дело в том, что активные секции имеют большую вероятность оказаться в кэше РСУБД в оперативной памяти, так как запросы к ним поступают чаще. А неактивные секции будут сохраняться лишь на диске, освобождая оперативную память под данные активных секций

Рассмотрим пример: БД журналов операций некоторого сервиса. Все действия пользователей записываются в БД для последующего отображения истории операций за текущий месяц, предыдущий месяц, квартал, полугодие, год. Если бизнес логика сервиса подразумевает, что пользователи чаще всего изучают свои операции за текущий и предыдущий месяцы, для такой БД можно реализовать секционирование по диапазонам значений - по месяцам. Операции создания идут только в секцию текущего месяца. Тогда индекс таблицы в начале месяца имеет нулевой размер и растет не бесконечно, а лишь до заключительного дня месяца, после чего создается уже новый индекс на следующий месяц. Секции предыдущих месяцев обрабатывают лишь редкие операции чтения.

В такой ситуации таблицы текущего и предыдущего месяцев будут являться активными сегментами, а все предыдущие таблицы - неактивными. И если для РСУБД будет выделен двукратный объем оперативной памяти соразмерный ежемесячной таблице, то все популярные запросы пользователей будут обрабатываться без использования диска. В итоге, взамен огромной таблицы, содержащей записи за многие годы, большинство которых на самом деле никому не интересны, секционирование позволяет держать в оперативной памяти две таблицы с индексами небольшого размера, содержащие действительно необходимые для пользователей нашего сервиса записи.

Если секции хранить в разных база данных или хранилищах на удаленных серверах - это *горизонтальное секционирование*, которое принято называть *сегментированием* (sharding). Рассмотрим сегментирование в следующем разделе

Сегментирование

Сегментирование (sharding) - это секционирование данных с размещением секций на нескольких серверах т.н. *шардах*. Подход впервые был опубликован ещё в 1988 году в статье Overview of SHARD: A System for Highly Available Replicated Data. Исходный текст данной статьи сложно найти в интернете, но она есть в некоторых библиотеках. В статье описывается подход компании Computer Corporation of America к созданию распределенных отказоустойчивых хранилищ данных

В разделе “Секционирование данных” указано, что вертикальное секционирование приносит позитивный эффект при проявлении активных секций. Существенное отличие сегментирования заключается в том, что оно будет полезным и когда все секции данных равновероятно активны, потому что на каждую секцию выделяется сервер, обладающей собственной подсистемой: процессором, памятью, диском и т.д.

Кроме повышенного требования к вычислительным мощностям, к недостатком сегментирования относятся сложности управления ввиду распределенного характера такой системы хранения данных:

- Плановое удаление шардов требует соответствующего переноса данных на оставшиеся таким образом, чтобы эти данные было возможно найти
- Агрегированное чтение, обновление, удаление данных, затрагивающее несколько шардов, требует использования сложных алгоритмов распределенных вычислений
- Проблема соответствия системы хранения данных принципам ACID

Рассмотрим пример: БД некоторой социальной сети, содержащая данные профилей пользователей. В качестве критерия сегментирования выберем список значений. Профили пользователей будут разделяться между тремя серверами `l0.db.local`, `l1.db.local` и `l3.db.local` по некоторым спискам: 0, 3, 6, 9... 1, 4, 7, 10… и 2, 5, 8, 11... Поиск шарда тогда происходит использованием простейшей хэш-функции остатка от деления (формула 3)

$$shardId = userId\%3 \quad (3)$$

Предположим, что по некоторой причине нам нужно будет избавиться от одного из серверов. В такой ситуации формула определения шарда изменится существенно (формула 4)

$$shardId = userId\%2 \quad (4)$$

Данная операция приведет к необходимости переноса 66% строк в системе хранения данных между хранилищами (см. таблицу 1)

Таблица 1

userId	shardId = userId%3	shardId = userId%2
0	l0.db.local	l0.db.local
1	l1.db.local	l1.db.local
2	l2.db.local	l0.db.local
3	l0.db.local	l1.db.local
4	l1.db.local	l0.db.local
5	l2.db.local	l1.db.local
6	l0.db.local	l0.db.local
7	l1.db.local	l1.db.local
8	l2.db.local	l0.db.local
9	l0.db.local	l1.db.local
10	l1.db.local	l0.db.local
11	l2.db.local	l1.db.local
12	l0.db.local	l0.db.local

Рассмотрим другой критерий сегментирования - диапазон значений: профили пользователей с идентификаторами 1-999 попадают на шард `d0.db.local`, 1000-1999 на шард `d1.db.local` и т.п. В такой ситуации с ростом социальной сети потребуется добавление новых шардов. Поиск шарда, на

котором находятся данные пользователя будет осуществляться функцией целочисленного деления (формула 5)

$$shardId = userId/1000 \quad (5)$$

Предположим, нам нужно будет избавиться от сервера `d4.db.local`. В такой ситуации всё решение с алгоритмической точки зрения перестает работать, т.к. нельзя идентификатор профиля, допустим, 4356 поделить на 1000 таким образом, чтобы получилось не 4, а 3 или 5. Проблему можно решить переносом данных на одно из хранилищ, например, на `d3.db.local` и созданием DNS-записи CNAME для `d4.db.local`, указав на `d3.db.local`. Нельзя сказать, что данное решение обладает изысканной простотой и элегантностью, но всё-таки лучше предыдущей схемы распределения профилей пользователей

Для более элегантного решения проблемы в 1997 году Дэвидом Каргером в соавторстве был предложен алгоритм консистентного хэширования (consistent hashing). С одной стороны он совмещает алгоритмическую чистоту решения, с другой не требует столь сильного переноса данных в случае удаления шардов. По некоторым оценкам в такой ситуации требуется перенос лишь 10-40% строк вместо 66% (см. таблицу 1). Данный подход очень популярен в настоящее время. Он реализовал в клиентах memcached и redis

Агрегированные операции чтения, изменения и удаления данных нескольких шардов решается при помощи алгоритмов Fork/Join и Map/Reduce. Алгоритм Fork/Join был предложен ещё в 1963 году в статье A multiprocessor system design. Map/Reduce был описан компанией Google в статье MapReduce: Simplified Data Processing on Large Clusters и является в некоторой степени упрощением Fork/Join в том плане, что он лишен возможности взаимодействия параллельных процессов друг с другом. И тот, и другой алгориты предполагаю разбитие исходной задачи на подзадачи, затем параллельное их выполнение, и, по завершении, сборка итогового результата (рисунок 4)

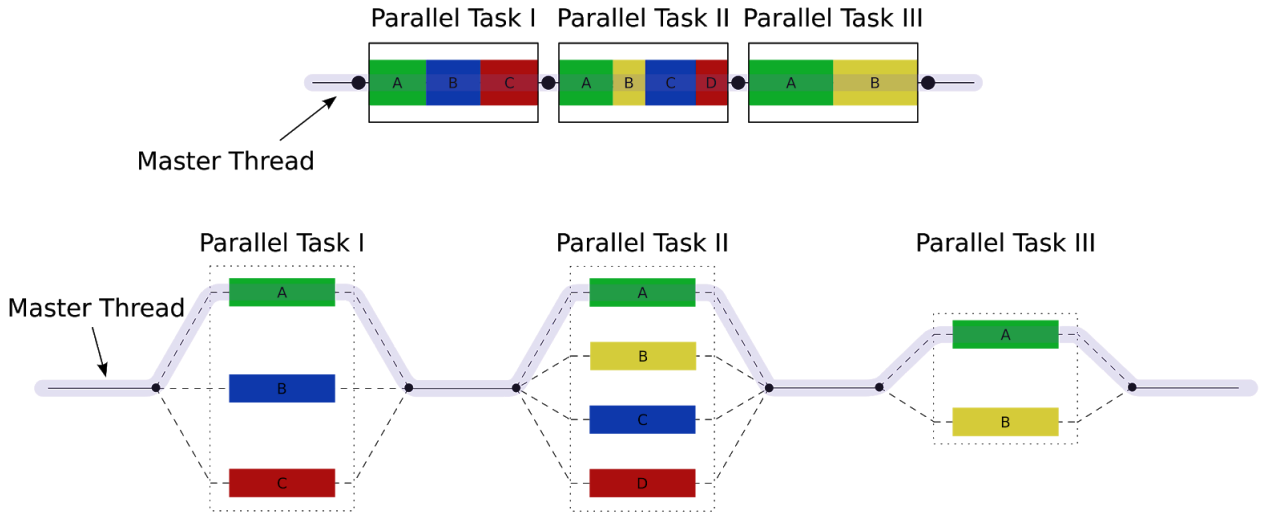

Рисунок 4 (*источник* - статья Fork/Join)

Если вернуться к примеру БД социальной сети с задачей выборки всех пользователей 30-летнего возраста, то это выполняется запуском SQL запроса параллельно на всех шардах:

```
SELECT "user_id" FROM "users" WHERE "age" BETWEEN 30 AND 39
```

Полученные записи собираем в один итоговый массив и возвращаем клиенту. Код данного решения на языке Golang приведен ниже:

```
var (
  done = make(chan struct{})
  res = make(chan int)
)
go func() {
  for userId := range res {
     fmt.Println(userId)
  }
  close(done)
}()

wg := &sync.WaitGroup{}
for _, shard = range shards {
  wg.Add(1)
  go func(shard *sharding.Shard) {
     defer wg.Done()
     rows, err := shard.Query(`SELECT "user_id" FROM "users" WHERE "age"
BETWEEN 30 AND 39`)
     if err != nil {
        //@todo process error
        return
     }
     defer rows.Close()

     for rows.Next() {
        var userId int
        err := rows.Scan(&userId)
        if err != nil {
           //@todo process error
           return
        }
        res <- userId
     }
  }(shard)
}
wg.Wait()
<-done
```

Особое внимание нужно уделить проблеме соблюдения ACID при использования сегментированных распределенных РСУБД. Предположим, возникла задача установления некоторого отношения между двумя пользователями в БД социальной сети: в том числе благодаря нашему сервису, пользователь с идентификатором **23453** вышла замуж за пользователя с идентификатором **45324** и мы, как разработчики, должны консистентно это зафиксировать. Два запроса должны быть выполнены на различных физических СУБД параллельно:

```
shard23.db.local: UPDATE users SET spouse=45324 WHERE user_id=23453

shard45.db.local: UPDATE users SET spouse=23453 WHERE user_id=45324
```

Очевидно, что просто запустив данные запросы параллельно, можно получить некорректное состояние. Один из запросов может выполниться успешно, но второй вернет ошибку или просто не выполнится по какой-либо причине - консистентность будет нарушена. Состояние профилей пользователей в социальной сети станет вызывать вопросы и, как ранее наш сервис способствовал созданию новой ячейки общества, он же может способствовать и ее разрушению, что, конечно, не хорошо.

Решается данная проблема использованием *распределенных транзакций*. Одним из самых известных алгоритмом реализации распределенной транзакции является двухфазный коммит, впервые описанный сотрудником компании Microsoft Филиппом Аланом Бернштейном. Подробнее с описанием алгоритма можно ознакомиться в статье на сайте Microsoft Docs

Следует отметить, что три указанных недостатка уже решены во многих СУБД - напр., в ElasticSearch, Aerospike. Но данные СУБД специализированны под специфические задачи. ElasticSearch используют в задачах быстрого сквозного поиска данных. Он не подойдет, если у вас присутствуют некоторые реляционные отношения в схеме данных. Aerospike является KV-хранилищем. РСУБД PostgreSQL обладает технологией Foreign Data Wrapper, позволяя сохранять секции данных на удаленных серверах. Но эта технология является относительно новой и ее внедрение следует осуществлять с предварительным тестированием. Аналогичную функциональность предлагают и облачные СУБД: Google Cloud Spanner, Amazon DynamoDB и т.п. Несмотря на многообразие готовых решений, на данный момент серверному программисту всё-таки лучше быть готовым к реализации сегментирования собственными силами под конкретную техническую задачу

Практика оптимизации систем хранения данных

В теоретической части урока мы рассмотрели подходы горизонтального масштабирования систем хранения данных. На практике по-отдельности они используются редко. Чаще происходит их совместное использование: например, сегментирование с секционированием таблицы внутри шарда и **master-slave** репликацией данных сегментов на резервные шарды (рисунок 5)

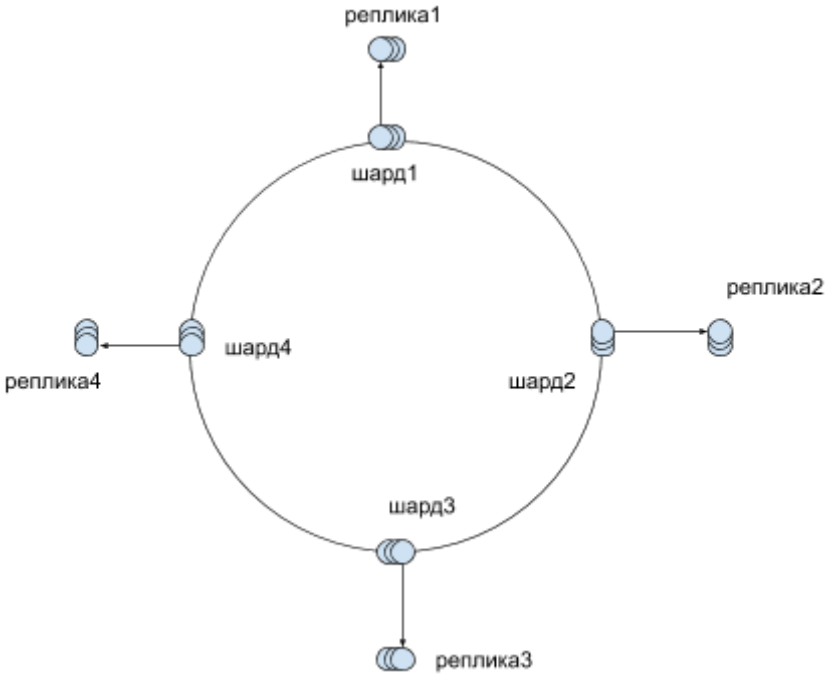

Рисунок 5

Реализуем систему хранения данных БД социальной сети, упомянутой ранее в разделе “Сегментирование”. В качестве подсистемы будем использовать РСУБД PostgreSQL. Пусть социальная сеть хранит профили пользователей и их активность. Опишем две таблицы: `users`, `activities`:

```
CREATE TABLE "users" (
    "user_id" INT,
    "name" VARCHAR,
    "age" INT,
    "spouse" INT
);
CREATE UNIQUE INDEX "users_user_id" ON "users" ("user_id");

CREATE TABLE "activities" (
    "user_id" INT,
    "date" TIMESTAMP,
    "name" VARCHAR
);
CREATE INDEX "activities_user_id_date" ON "activities" ("user_id", "date");
```

Реализацию системы хранения данных начнем последовательно, с репликации

Репликация БД

Для создания репликации для таблиц воспользуется готовым решением от компании Bitnami, которая подготовила образы PostgreSQL, поддерживающие основные настройки репликации через переменные окружения. Опишем нашу инфраструктуру при помощи Docker Compose:

```
master:
 image: docker.io/bitnami/postgresql:12.5.0
```

```
    ports:
      - 8081:5432
    environment:
      - POSTGRESQL_REPLICATION_MODE=master
      - POSTGRESQL_REPLICATION_USER=replicator
      - POSTGRESQL_REPLICATION_PASSWORD=secret_password
      - POSTGRESQL_USERNAME=test
      - POSTGRESQL_PASSWORD=test
      - POSTGRESQL_DATABASE=test
      - ALLOW_EMPTY_PASSWORD=yes

  slave:
    image: docker.io/bitnami/postgresql:12.5.0
    ports:
      - 8082:5432
    depends_on:
      - master
    environment:
      - POSTGRESQL_REPLICATION_MODE=slave
      - POSTGRESQL_REPLICATION_USER=replicator
      - POSTGRESQL_REPLICATION_PASSWORD=secret_password
      - POSTGRESQL_MASTER_HOST=master
      - POSTGRESQL_MASTER_PORT_NUMBER=5432
      - POSTGRESQL_PASSWORD=test
      - ALLOW_EMPTY_PASSWORD=yes
```

Если запустить данную инфраструктуру, то в хранилище **slave** начнут копироваться все изменения хранилища **master**, используя потоковую репликацию записей WAL. Выполните команду `docker-compose up -d`. Выполнив `docker ps` можно убедиться, что оба хранилища РСУБД запущены и готовы принимать соединения:

```
$: docker ps
CONTAINER ID        IMAGE                          COMMAND
CREATED             STATUS              PORTS                      NAMES
07db679650d3        bitnami/postgresql:12.5.0   "/opt/bitnami/script…"   3
minutes ago       Up 3 minutes        0.0.0.0:8082->5432/tcp   slave_1
bbc136b494ce        bitnami/postgresql:12.5.0   "/opt/bitnami/script…"   3
minutes ago       Up 3 minutes        0.0.0.0:8081->5432/tcp   master_1
```

Проверим репликацию выполнением **DDL запроса** на создание таблицы `users`:

```
goose postgres 'port=8081 user=test password=test dbname=test
sslmode=disable' up
2020/11/30 19:09:21 OK    20201130182952_Users.sql
2020/11/30 19:09:21 goose: no migrations to run. current version:
20201130182952
```

Проверим, что обе РСУБД: и **master** и **slave** - теперь содержат таблицу users:

```
$: psql --host=127.0.0.1 --port=8081 --username=test -c 'select * from
users'
 user_id | age | spouse
---------+-----+--------
(0 rows)
```

```
$: psql --host=127.0.0.1 --port=8082 --username=test -c 'select * from
users'
 user_id | age | spouse
---------+-----+--------
(0 rows)
```

Проверим репликацию **DML-запроса** на создание новой строки данных в хранилище **master**. Обратите внимание, запись производится в хранилище **master**, а чтение из хранилища **slave**:

```
$: psql --host=127.0.0.1 --port=8081 --username=test -c "insert into users
values (1, 'Joe Biden', 78, 0)"
INSERT 0 1

$: psql --host=127.0.0.1 --port=8082 --username=test -c 'select * from
users'
 user_id |   name    | age | spouse
---------+-----------+-----+--------
       1 | Joe Biden |  78 |      0
(1 row)
```

В результате выполнения SQL-запроса, видно, что данные были реплицированы и на хранилище **slave** успешно. Таким образом, мы решили проблему репликации данных на резервный сервер, на который, кроме прочего, можно отправлять и операции чтения данных

Секционирвоание таблиц

Предполагается, что таблица активностей пользователей будет постоянно накапливать данные. Размер таблицы будет неуклонно расти. Настроим секционирование данных для таблицы `activities` по месяцам. При таком решении при выборке данных в кэш РСУБД, в оперативной памяти, будут попадать активные секции. т.е. секции с данными наиболее популярных месяцев - ориентировочно, текущего и предыдущего. Обновим миграцию для настройки секционирования таблицы:

```
CREATE TABLE "activities" (
   "user_id" INT,
   "date" TIMESTAMP,
   "name" VARCHAR
) PARTITION BY RANGE("date");
CREATE INDEX "activities_user_id_date" ON "activities" ("user_id", "date");
CREATE TABLE "activities_202011" PARTITION OF "activities" FOR VALUES FROM
('2020-11-01'::TIMESTAMP) TO ('2020-12-01'::TIMESTAMP);
CREATE TABLE "activities_202012" PARTITION OF "activities" FOR VALUES FROM
('2020-12-01'::TIMESTAMP) TO ('2021-01-01'::TIMESTAMP);
```

Теперь данные по активностям пользователей в ноябре и декабре 2020 года будут попадать в выделенные секции. Проверим трудозатраты РСУБД на выполнение запросов:

```
$: psql --host=127.0.0.1 --port=8082 --username=test -c "explain select *
from activities where user_id=1 and date='2020-11-01'::timestamp"
Password for user test:
                                                 QUERY PLAN
------------------------------------------------------------------------------
--------------------------------
```

```
 Index Scan using activities_202011_user_id_date_idx on activities_202011
(cost=0.15..8.17 rows=1 width=44)
   Index Cond: ((user_id = 1) AND (date = '2020-11-01 00:00:00'::timestamp
without time zone))
(2 rows)
```

Вы можете видеть, что для выборки данных за 01 ноября 2020 года планируется использование секции, несмотря на то, что мы написали запрос на выборку из таблицы `activities` в целом. Попробуем оценить выборку всех данных для заданного пользователя:

```
$: psql --host=127.0.0.1 --port=8082 --username=test -c "explain select *
from activities where user_id=1"
Password for user test:
                                                QUERY PLAN
-----------------------------------------------------------------------------
---------------------------
 Append  (cost=4.20..27.39 rows=12 width=44)
   ->  Bitmap Heap Scan on activities_202011  (cost=4.20..13.67 rows=6
width=44)
         Recheck Cond: (user_id = 1)
         ->  Bitmap Index Scan on activities_202011_user_id_date_idx
(cost=0.00..4.20 rows=6 width=0)
               Index Cond: (user_id = 1)
   ->  Bitmap Heap Scan on activities_202012  (cost=4.20..13.67 rows=6
width=44)
         Recheck Cond: (user_id = 1)
         ->  Bitmap Index Scan on activities_202012_user_id_date_idx
(cost=0.00..4.20 rows=6 width=0)
               Index Cond: (user_id = 1)
(9 rows)
```

Запрос требует сканирования нескольких секционированных индексов через Bitmap Scan, т.к. мы запросили много строк, но не все, а только с `user_id=1`

Таким образом, мы настроили секционирование таблицы активностей и теперь можем производить выборку данных быстрее, обращаясь преимущественно к активным секциям. План выполнения запроса показывает уменьшение времени их выполнения от $T(n)$ к, фактически, $T(\frac{n}{m})$, где m - число секций

Сегментирование БД

Система хранения данных, разработанная нами на данный момент, уже является доступной, за счет разделения операций чтения, обновления, создания и удаления данных, и надежной за счет наличия резервного копирования на дополнительные хранилища. Для оценки масштабируемости предположим, что произойдет, если общее число пользователей составить $100 * 10^6$, а показатель MAU будет $10 * 10^6$:

- Таблица `users` будет иметь $100 * 10^6$ строк. Индекс `users_user_id` будет представлять собой дерево с $100 * 10^6$ узлами

- Секция текущего месяца таблицы `activities` будет иметь $(10 * n) * 10^6$ записей, где n - среднее число активной пользователя в месяц. Индекс `activities_user_id_date` будет иметь $10 * 10^6$ узлов, каждый из которых в среднем будет иметь поддерево с n узлами

Итерационно вставим несколько тысяч строк данных в наша хранилище, чтобы приблизительно оценить рост потребления ресурсов для хранения столь больших индексов:

```
$: for USER in {1..1000}; do PGPASSWORD=test psql --host=127.0.0.1
--port=8081 --username=test -c "insert into users values ($USER, '', 18,
0)"; done
$: for USER in {1001..2000}; do PGPASSWORD=test psql --host=127.0.0.1
--port=8081 --username=test -c "insert into users values ($USER, '', 18,
0)"; done
$: for USER in {2001..3000}; do PGPASSWORD=test psql --host=127.0.0.1
--port=8081 --username=test -c "insert into users values ($USER, '', 18,
0)"; done
```

Оценим размер индексов по данным служебной таблицы `pg_class`:

```
$: psql --host=127.0.0.1 --port=8081 --username=test -c "ANALYZE users;"
ANALYZE
$: psql --host=127.0.0.1 --port=8081 --username=test -c "SELECT
sum(relpages*BLCKSZ) FROM pg_class WHERE reltype=0 and
relname='users_user_id';"
 sum
-----
  40
(1 row)
```

Здесь `relpages` - размер представления в блоках, а `BLCKSZ` по-умолчанию составляет **8Кб**

Выполнив измерения можно заметить что на каждую тысячу пользователей индекс таблицы users прирастает на **24Кб**. Таким образом, для хранения $100 * 10^6$ пользователей нам понадобится примерно **2.2Тб** памяти. Чтобы иметь возможность подобного масштабирования нашей системы хранения данных, как было отмечено в разделе “Теория урока”, мы должны отказаться от идеи вертикального масштабирования и выполнить масштабирование горизонтальное, чем и является сегментирование применительно к РСУБД.

Напишем простейший менеджер, который сегментирует данные по диапазону значений:

```
type Shard struct {
  Address string
  Number    int
}

type Manager struct {
  size int
  ss  *sync.Map
}

var (
  ErrorShardNotFound = errors.New("shard not found")
```

```
)

func NewManager(size int) *Manager {
  return &Manager{
     size: size,
     ss: &sync.Map{},
  }
}

func (m *Manager) Add(s *Shard) {
  m.ss.Store(s.Number, s)
}

func (m *Manager) ShardById(entityId int) (*Shard, error) {
  if entityId < 0 {
     return nil, ErrorShardNotFound
  }
  n := entityId/m.size
  if s, ok := m.ss.Load(n); ok {
     return s.(*Shard), nil
  }
  return nil, ErrorShardNotFound
}
```

Теперь реализуем небольшой пул соединений к нашим шардам, чтобы не переоткрывать их регулярно:

```
type Pool struct {
  sync.RWMutex

  cc map[string]*sql.DB
}

func NewPool() *Pool {
  return &Pool{
     cc: map[string]*sql.DB{},
  }
}

func (p *Pool) Connection(addr string) (*sql.DB, error) {
  p.RLock()
  if c, ok := p.cc[addr]; ok {
     defer p.RUnlock()
     return c, nil
  }
  p.RUnlock()

  p.Lock()
  defer p.Unlock()
  if c, ok := p.cc[addr]; ok {
     return c, nil
  }
  var err error
  p.cc[addr], err = sql.Open("postgres", addr)
  return p.cc[addr], err
}
```

Добавим шарды в описание нашей инфраструктуры:

```
shard_0:
 image: docker.io/bitnami/postgresql:12.5.0
 ports:
   - 8100:5432
 environment:
   - POSTGRESQL_REPLICATION_MODE=master
   - POSTGRESQL_REPLICATION_USER=replicator
   - POSTGRESQL_REPLICATION_PASSWORD=secret_password
   - POSTGRESQL_USERNAME=test
   - POSTGRESQL_PASSWORD=test
   - POSTGRESQL_DATABASE=test
   - ALLOW_EMPTY_PASSWORD=yes

replica_0:
 image: docker.io/bitnami/postgresql:12.5.0
 ports:
   - 8101:5432
 depends_on:
   - shard_0
 environment:
   - POSTGRESQL_REPLICATION_MODE=slave
   - POSTGRESQL_REPLICATION_USER=replicator
   - POSTGRESQL_REPLICATION_PASSWORD=secret_password
   - POSTGRESQL_MASTER_HOST=shard_0
   - POSTGRESQL_MASTER_PORT_NUMBER=5432
   - POSTGRESQL_PASSWORD=test
   - ALLOW_EMPTY_PASSWORD=yes

shard_1:
 image: docker.io/bitnami/postgresql:12.5.0
 ports:
   - 8110:5432
 environment:
   - POSTGRESQL_REPLICATION_MODE=master
   - POSTGRESQL_REPLICATION_USER=replicator
   - POSTGRESQL_REPLICATION_PASSWORD=secret_password
   - POSTGRESQL_USERNAME=test
   - POSTGRESQL_PASSWORD=test
   - POSTGRESQL_DATABASE=test
   - ALLOW_EMPTY_PASSWORD=yes

replica_1:
 image: docker.io/bitnami/postgresql:12.5.0
 ports:
   - 8111:5432
 depends_on:
   - shard_1
 environment:
   - POSTGRESQL_REPLICATION_MODE=slave
   - POSTGRESQL_REPLICATION_USER=replicator
   - POSTGRESQL_REPLICATION_PASSWORD=secret_password
   - POSTGRESQL_MASTER_HOST=shard_1
   - POSTGRESQL_MASTER_PORT_NUMBER=5432
   - POSTGRESQL_PASSWORD=test
   - ALLOW_EMPTY_PASSWORD=yes

shard_2:
```

```
    image: docker.io/bitnami/postgresql:12.5.0
    ports:
      - 8120:5432
    environment:
      - POSTGRESQL_REPLICATION_MODE=master
      - POSTGRESQL_REPLICATION_USER=replicator
      - POSTGRESQL_REPLICATION_PASSWORD=secret_password
      - POSTGRESQL_USERNAME=test
      - POSTGRESQL_PASSWORD=test
      - POSTGRESQL_DATABASE=test
      - ALLOW_EMPTY_PASSWORD=yes

  replica_2:
    image: docker.io/bitnami/postgresql:12.5.0
    ports:
      - 8121:5432
    depends_on:
      - shard_2
    environment:
      - POSTGRESQL_REPLICATION_MODE=slave
      - POSTGRESQL_REPLICATION_USER=replicator
      - POSTGRESQL_REPLICATION_PASSWORD=secret_password
      - POSTGRESQL_MASTER_HOST=shard_2
      - POSTGRESQL_MASTER_PORT_NUMBER=5432
      - POSTGRESQL_PASSWORD=test
      - ALLOW_EMPTY_PASSWORD=yes
```

Создадим небольшую модель данных, использующую ранее созданный пул соединений к сегментам РСУБД:

```
type User struct {
  UserId int
  Name string
  Age int
  Spouse int
}

func (u *User) connection() (*sql.DB, error) {
  s, err := m.ShardById(u.UserId)
  if err != nil {
     return nil, err
  }
  return p.Connection(s.Address)
}
```

Функция `connection` обеспечивает поиск шарда, необходимого для указанного экземпляра модели. Используя данную функцию уже возможно реализовать непосредственно CRUD-интерфейс модели `User`:

```
func (u *User) Create() error {
  c, err := u.connection()
  if err != nil {
     return err
  }
  _, err = c.Exec(`INSERT INTO "users" VALUES ($1, $2, $3, $4)`, u.UserId,
u.Name, u.Age, u.Spouse)
  return err
```

```
}

func (u *User) Read() error {
  c, err := u.connection()
  if err != nil {
     return err
  }
  r := c.QueryRow(`SELECT "name", "age", "spouse" FROM "users" WHERE
"user_id" = $1`, u.UserId)
  return r.Scan(
     &u.Name,
     &u.Age,
     &u.Spouse,
  )
}

func (u *User) Update() error {
  c, err := u.connection()
  if err != nil {
     return err
  }
  _, err = c.Exec(`UPDATE "users" SET "name" = $2, "age" = $3, "spouse" =
$4 WHERE "user_id" = $1`, u.UserId,
     u.Name, u.Age, u.Spouse)
  return err
}

func (u *User) Delete() error {
  c, err := u.connection()
  if err != nil {
     return err
  }
  _, err = c.Exec(`DELETE FROM "users" WHERE "user_id" = $1`, u.UserId)
  return err
}
```

Выполним код создания нескольких пользователей на описанной ранее инфраструктуре:

```
func main()  {
  m.Add(&sharding.Shard{"port=8100 user=test password=test dbname=test
sslmode=disable", 0})
  m.Add(&sharding.Shard{"port=8110 user=test password=test dbname=test
sslmode=disable", 1})
  m.Add(&sharding.Shard{"port=8120 user=test password=test dbname=test
sslmode=disable", 2})

  uu := []*User{
     {1, "Joe Biden", 78, 10},
     {10, "Jill Biden", 69, 1},
     {13, "Donald Trump", 74, 25},
     {25, "Melania Trump", 78, 13},
  }
  for _, u := range uu {
     err := u.Create()
     if err != nil {
        fmt.Println(fmt.Errorf("error on create user %v: %w", u, err))
     }
  }
}
```

Чтобы проверить, что и код, и инфраструктура работают как ожидается, выполняем SQL запросы к хранилищам **slave**, а не к исходным шардам, к которым были созданы соединения:

```
$: psql --host=127.0.0.1 --port=8101 --username=test -c "SELECT * FROM
users;"
 user_id |   name    | age | spouse
---------+-----------+-----+--------
       1 | Joe Biden |  78 |     10
(1 row)

$: psql --host=127.0.0.1 --port=8111 --username=test -c "SELECT * FROM
users;"
 user_id |     name     | age | spouse
---------+--------------+-----+--------
      10 | Jill Biden   |  69 |      1
      13 | Donald Trump |  74 |     15
(2 rows)

$: psql --host=127.0.0.1 --port=8121 --username=test -c "SELECT * FROM
users;"
 user_id |     name      | age | spouse
---------+---------------+-----+--------
      25 | Melania Trump |  78 |     10
(1 row)
```

Таким образом, мы реализовали структуру БД, представленную на рисунке 5. Мы добавили сегменты для распределения данных пользователей для обеспечения масштабирования системы хранения данных. Таблицы с высоким процентом архивных данных - активностей пользователя - мы разбили на секции средствами PostgreSQL. Зафиксировали надежность и доступность системы хранения данных, реализовав **master-slave** репликацию

Практические задания

Для закрепления полученных знаний, выполните задания:

1. Создайте модель на языке Golang для доступа к таблице активностей с использованием менеджера шардов по аналогии с таблицей профилей пользователей
2. Модифицируйте обе модели таким образом, чтобы лучше использовать реализованный механизм репликации данных: запись изменений нужно осуществлять на **master**, а чтение данных можно выполнять с хранилища **slave**
3. Модифицируйте решение полученное в задании №2 таким образом, чтобы чтение данных происходило не только с хранилища **slave**, но и из хранилища **master** равновероятно
4. Самостоятельно реализуйте менеджер шардов с использованием алгоритма консистентного хэширования. Для выполнения задания можно подсмотреть как это сделали разработчики библиотеки github.com/go-redis/redis. Дополнительно вам может помочь статья Consistent Hashing: Algorithmic Tradeoffs

Используемые источники

1. Перевод классической книги Никлауса Вирта “Алгоритмы + структуры данных = программы”: https://www.litres.ru/niklaus-virt/algoritmy-i-struktury-dannyh-22072427/
2. Практическое сравнение OLTP и OLAP систем: https://www.stitchdata.com/resources/oltp-vs-olap/
3. Визуализация деревьев как структур данных: http://btv.melezinek.cz/
4. Статья “О-больше и о-малое”: https://ru.wikipedia.org/wiki/%C2%ABO%C2%BB_%D0%B1%D0%BE%D0%BB%D1%8C%D1%88%D0%BE%D0%B5_%D0%B8_%C2%ABo%C2%BB_%D0%BC%D0%B0%D0%BB%D0%BE%D0%B5
5. Исследование производительности системного вызова `fsync` от компании Percona: https://www.percona.com/blog/2018/02/08/fsync-performance-storage-devices
6. Законы Мура: https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%BA%D0%BE%D0%BD_%D0%9C%D1%83%D1%80%D0%B0
7. Статья с описание алгоритма консистентного хэширования: https://dl.acm.org/doi/10.1145/258533.258660
8. Статья с описанием алгоритма Fork/Join: https://dl.acm.org/doi/10.1145/1463822.1463838
9. Статья с описанием алгоритма Map/Reduce: https://research.google/pubs/pub62/
10. Статья с описание алгоритма двухфазного коммита: https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-tpsod/f048d118-9a8f-4300-abca-9333204acd75