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

Факультет компьютерных наук
Основная образовательная программа
Прикладная математика и информатика

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

Программный проект
на тему
RAID в Userspace для ClickHouse

Выполнил студент гр. **БПМИ165**, 4 курса,
Новиков Глеб Александрович

Руководитель ВКР:
Доцент, М базовая кафедра Яндекс,
Миловидов Алексей Николаевич

Abstract

ClickHouse is analitycal open-source column-oriented database management system. One of its key features is performance of aggregating read requests with tons of data. RAID — redundant array of independent disks, is the most popular way to increase performance and storage volume of local storage, which is provided by effective usage of multiple disks in different ways (RAID 0, RAID 1, RAID 5 etc). Implementation of RAID in userspace of application gives a lot of agility and application-specific advantages to application developers and users. In this paper we describe existing RAID types, investigate ClickHouse architecture, MergeTree data storage logical and software architecture and describe an implementation of RAID in userspace for ClickHouse, leaving agile architecture in codebase, which allows to implement any other RAID type and improve replicas restoring procedure of MergeTree distributed tables.

Аннотация

ClickHouse это аналитическая колоночно-ориентированная система управления базами данных с открытым исходным кодом. Одним из основных преимуществ ClickHouse является его производительность на аналитических запросов, выполняющихся с анализом больших объёмов данных. RAID — технология виртуализации данных для объединения нескольких дисковых устройств, является одним из самых популярных способов увеличить производительность и объём компьютерной системы с помощью добавления дисков в различных конфигурациях, таких как RAID 0, RAID 1, RAID 5 и другие. Реализация RAID в userspace для какого-либо приложения даёт большую гибкость разработчикам и пользователям приложения, давая возможность не только эффективно работать с персистентными данными приложения, но и завязав на несколько дисков логику каких-то процессов внутри приложения. В этой работе мы описываем существующие типы RAID, исследуем архитектуру хранения данных в MergeTree (семейство типов таблиц ClickHouse), исследуем и описываем программную реализацию механизмов хранения данных в MergeTree, а так же изменяем программную архитектуру и реализуем прототип RAID в кодовой базе ClickHouse. В дальнейшем подготовленная архитектура позволит реализовать другие типы RAID в кодовой базе ClickHouse, а так же усовершенствовать процедуры восстановления реплик распределённых таблиц семейства MergeTree.

1 Введение

ClickHouse — аналитическая база данных с открытым исходным кодом. Основные пользовательские сценарии использования ClickHouse это хранение сырых событий с довольно большим количеством колонок и аналитические запросы по этим событиям, например хранение DNS запросов или событий для таких систем, как Яндекс.Метрика или Google Analytics. ClickHouse позволяет хранить и обрабатывать терабайты сырых данных как на одном сервере, так и в распределенной системе серверов. Скорость аналитических запросов в ClickHouse является одним из его основных преимуществ.

Основным движком таблиц в ClickHouse является движок MergeTree (на самом деле это большое семейство движков). В основе MergeTree лежит идея классического LSM — Log Structured Mergetree: данные хранятся в неизменяемых отсортированных файлах, и периодически объединяются (в фоне) в отсортированные файлы побольше. Для изменения и удаления данных так же происходит слияние файлов, в результате которого будет уже новый обновлённый файл. Подробнее о том, как устроен MergeTree, написано в главе про его архитектуру Архитектура хранения данных MergeTree. Данные разделены на большие *партиции* (например, по месяцу записи), а партиции состоят из *кусков (parts)*, куски и являются теми самыми неизменяемыми файлами из LSM. Партиция может состоять из одного куска, а данные одной партиции могут весить очень много — сотни гигабайт или терабайты. Чтобы сделать хранение данных более надёжным, а конфигурацию более гибкой, предлагается реализовать классический подход к использованию нескольких дисков для хранения данных — RAID.

RAID (redundant array of independent disks) — набор дисков, объединённых в виртуальное дисковое пространство. Фактически, множество дисков, иммитирующее собой единое устройство. Эта концепция была сформулирована в 1988 году [4], когда рынок жёстких дисков состоял из медленных механических дисков, имеющих небольшие объёмы. Различные конфигурации RAID были призваны увеличить скорость, объём и/или надёжность хранилища засчёт добавления дисков в массив. Самые популярные конфигурации — RAID

1 и RAID 0. RAID 1 подразумевает полное зеркалирование действий с несколькими дисками, тем самым достигается отказоустойчивость при поломке одного из дисков (рис. 1). RAID 0 устроен несколько сложнее — записываемый файл бьётся на чанки по фиксированному количеству байт и пишется на диски друг за другом, для каждого следующего чанка меняя диск на следующий в массиве (пример есть на рис. 2). Это позволяет равномерно распределить данные на нескольких дисках, заполняя их постепенно. Более того, данные на нескольких дисках позволяют модифицировать процедуру чтения таким образом, чтобы читать сразу с нескольких дисков сразу (часто скорость жёстких дисков является бутылочным горлышком в производительности приложений).

Поддержка множества дисков в парадигме RAID может быть реализована на разных уровнях:

- можно разработать и собрать физический контроллер, в который включается несколько дисков, а процессор контроллера сам обрабатывает операции записи и чтения и выбирает, каким образом он будет выполнять операции над дисками;

- многие современные материнские платы поддерживают подключение нескольких жёстких дисков, поэтому можно реализовать поддержку RAID над несколькими дисками на уровне BIOS, таким образом обойдясь без дорогостоящего нестандартного оборудования, но воспользовавшись частью мощности основного CPU, что сделает RAID несколько медленнее, чем при использовании отдельного контроллера;

- можно разработать драйвер для операционной системы, который будет выступать некоторой прослойкой между файловой системой и жёсткими дисками, в этом случае все операции с дисками будут так же обрабатываться на основном процессоре, зато не придётся покупать дорогостоящие контроллеры или писать ПО на уровне BIOS;

- самым *верхним* с точки зрения разработки ПО является способ реализации взаимодействия с несколькими дисками прямо на уровне приложения,

то есть в файловой системе как угодно (FUSE, физические диски) появляется несколько дисков, приложению сообщаются точки монтирования и оно взаимодействует с ними как с разными дисками.

В рамках данной ВКР нас интересует самый последний способ реализации RAID — на уровне приложения, или, как принято говорить, в пользовательском пространстве или *в userspace*. Реализация RAID в userspace позволит ClickHouse иметь собственные настройки работы над несколькими дисками, избавляя от зависимостей в ОС. Более того, это даст возможность управлять работой с данными над несколькими дисками прямо в кодовой базе ClickHouse. Такая гибкость позволяет усовершенствовать некоторые процессы работы с данными — например, восстановление данных в реплике распределённой таблицы может происходить не с другой реплики, а с другого диска, в случае выхода из строя одного из дисков реплики, это позволит сэкономить на трафике и ускорить процесс восстановления реплики. Кроме того, подготовленная архитектура работы с данными позволит реализовать и другие подходы к работе с данными на диске, которые не поддерживаются утилитами ОС, такие как RAID-Z или RAID-1E.

В классическом RAID разбиение на чанки происходит по фиксированному объёму байт. В ClickHouse единицей записи куска является *гранула* — промежуток таблицы из последовательных строк. С недавнего времени все гранулы имеют близкий друг к другу размер (*adaptive granularity*), поэтому в нашем случае будет интереснее реализовать разбиение кусков по дискам на уровне гранул.

Сформулируем задачи, которые необходимо решить в рамках данной работы:

1. изучение возможных конфигураций RAID;
2. изучение концепций в хранении данных MergeTree;
3. подробное изучение и описание программной архитектуры процессов записи, чтения, слияния кусков;

4. описание и прототипирование изменений в процедурах работы с данными для поддержания как текущей функциональности, так и возможного расширения процессов для работы с несколькими дисками;

5. внесение изменений в существующую архитектуру процессов работы с данными;

6. описание и прототипирование RAID в рамках измененной архитектуры процессов работы с данными;

7. реализация прототипа RAID на подготовленной архитектуре;

В рамках этой работы у нас получилось успешно выполнить задачи 1-6. К сожалению, изначально объём задачи казался значительно меньше и за время работы над ВКР мы успели только подготовить код для реализации RAID, но не полноценно реализовать его. В главе *RAID* работы описано устройство различных конфигураций массивов дисков. В главе *Обзор архитектуры MergeTree в ClickHouse* подробно описана схема хранения данных MergeTree, а так же реализация записи, чтения и других механизмов работы с данными в MergeTree. В главе *Реализация RAID в коде ClickHouse* описаны изменения, которые были внесены в существующую архитектуру для поддержки RAID в коде MergeTree.

2 RAID

Как уже было сказано, существует множество различных конфигураций RAID с разными количествами дисков и разной логикой записи. Рассмотрим наиболее популярные из них.

2.1 Конфигурации RAID

2.1.1 Простейшие конфигурации

Механические диски имеют свойство часто выходить из строя, поэтому давайте воспользуемся первой идеей — будем производить все операции

с несколькими дисками одновременно. Таким образом, у нас получится несколько зеркальных дисков, при выходе одного из строя данные на всех остальных остаются целыми, можно добавить новый диск и скопировать на него содержимое любого другого диска. Эта конфигурация называется *RAID 1* и, как правило, использует два диска (рис. 1). При использовании множества дисков в RAID 1 общий объём хранилища равен минимальному объёму из всех дисков массива. Этот тип RAID не ускоряет запись на диск, а скорее даже замедляет — запись на все диски в лучшем случае (при параллельной записи) будет произведена со скоростью самого медленного диска в массиве. С другой стороны, чтение данных с диска можно осуществлять параллельно: пусть в массиве n дисков, узнаем размер файла $fileSize$, делим файл на n кусочков и параллельно читаем эти куски размера $\frac{fileSize}{n}$ с разных дисков. Таким образом, скорость чтения при использовании RAID 1 возрастает в n раз.

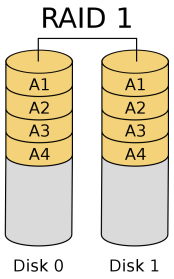


Figure 1: Пример RAID 1 над двумя дисками

Что делать, если мы уверены в своих жёстких дисках и нам надо расширить объём пространства с помощью добавления новых дисков? Воспользуемся следующей идеей: файлы можно бить на чанки какого-то небольшого размера и писать их последовательно на диски массива, для каждого следующего чанка переключая диск на следующий в массиве. Эта техника называется *striping* и в полной мере представлена в конфигурации RAID 0 (рис. 2).

RAID 0 позволяет распараллелить как запись, так и чтение, таким образом для массива из n дисков увеличив скорость в n раз. Однако, при

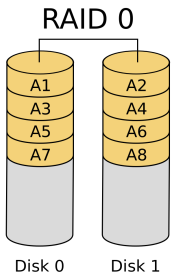


Figure 2: Пример RAID 0 над двумя дисками

выходе из строя одного из дисков, данные на всех дисках теряются, так как их недостающие куски лежали на вышедшем из строя диске.

Теперь естественным кажется совмещение RAID 0 и RAID 1 в конфигурацию из четырёх дисков. Действительно, конфигурации RAID 01 или RAID 10 являются одними из самых часто используемых. Идея очень простая — давайте возьмём по два диска в конфигурации RAID 0 или RAID 1 и построим над двумя парами как над виртуальными дисками RAID 1 или RAID 0 соответственно (рис. 3)

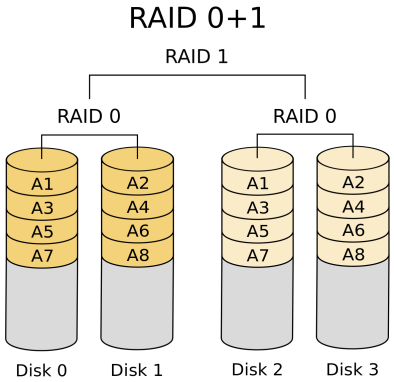


Figure 3: Пример RAID 01 над четырьмя дисками

Нетрудно заметить, что в RAID 01 из четырех дисков общий объём используется только наполовину. Для увеличения эффективности использования ёмкости дисков существуют более сложные конфигурации RAID.

2.1.2 RAID 4, RAID 5, RAID 6

Рассмотрим RAID 0. В нём нас многое устраивало — высокоэффективное использование доступной ёмкости, линейное увеличение скорости и объёма виртуального диска с ростом количества дисков в массиве. Единственное, чего нам не хватало — отказоустойчивости. Давайте выделим один диск в массиве и воспользуемся им для хранения информации, которая поможет в последствии восстановить данные на вышедшем из строя диске. В массиве из n дисков будем делить записываемый файл на $n-1$ чанков и для всех байт в разрезе (то есть каждого $\frac{fileSize}{n-1}$ байта) будем писать результат выполнения операции xor от этих байт на диск с информацией для восстановления. Такие блоки байт называются *parity blocks*. Таким образом получится, что при выходе одного диска из строя мы можем либо восстановить все утерянные байты с помощью вспомогательного диска и данных на других дисках, либо просто вычислить xor, если был поломан диск с вспомогательной информацией. Такая конфигурация называется RAID 4. Сейчас она не используется, так как следующая конфигурация является её более совершенным продолжением.

Вместо того, чтобы выделять отдельный диск для вспомогательной информации, для каждого нового слоя записи можно ротировать диск, на который будет записана вспомогательная информация. Таким образом схема восстановления едина для любого вышедшего из строя диска.

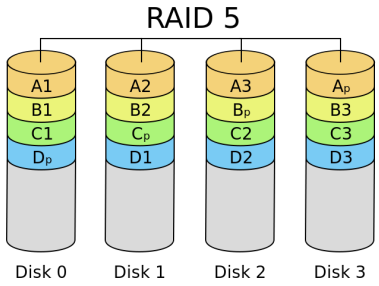


Figure 4: Пример RAID 5 над четырьмя дисками

Конфигурация RAID 5 позволяет получить линейный рост производительности

записи и чтения, а так же устойчивость к поломке одного диска в массиве. Всё это происходит в обмен на ёмкость всего лишь одного диска — то есть из n дисков объёма X для использования будет доступно $(n-1) \cdot X$.

Чтобы добавить отказоустойчивости в RAID 5, можно воспользоваться ещё одной, независимой от *xor* функцией для вычисления *parity block*. Результат вычисления этой функции будем писать на ещё один диск, таким образом из n дисков в массиве будет использовано $n-2$ диска в полном объёме. Такая конфигурация называется RAID 6 и тоже иногда используется — когда есть необходимость в большей отказоустойчивости, чем даёт RAID 5.

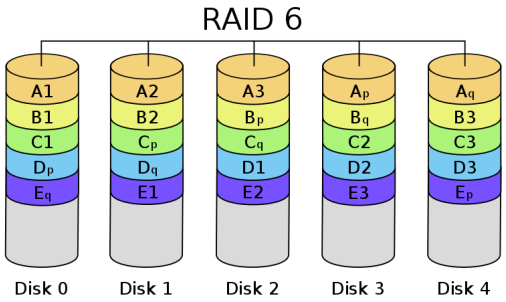
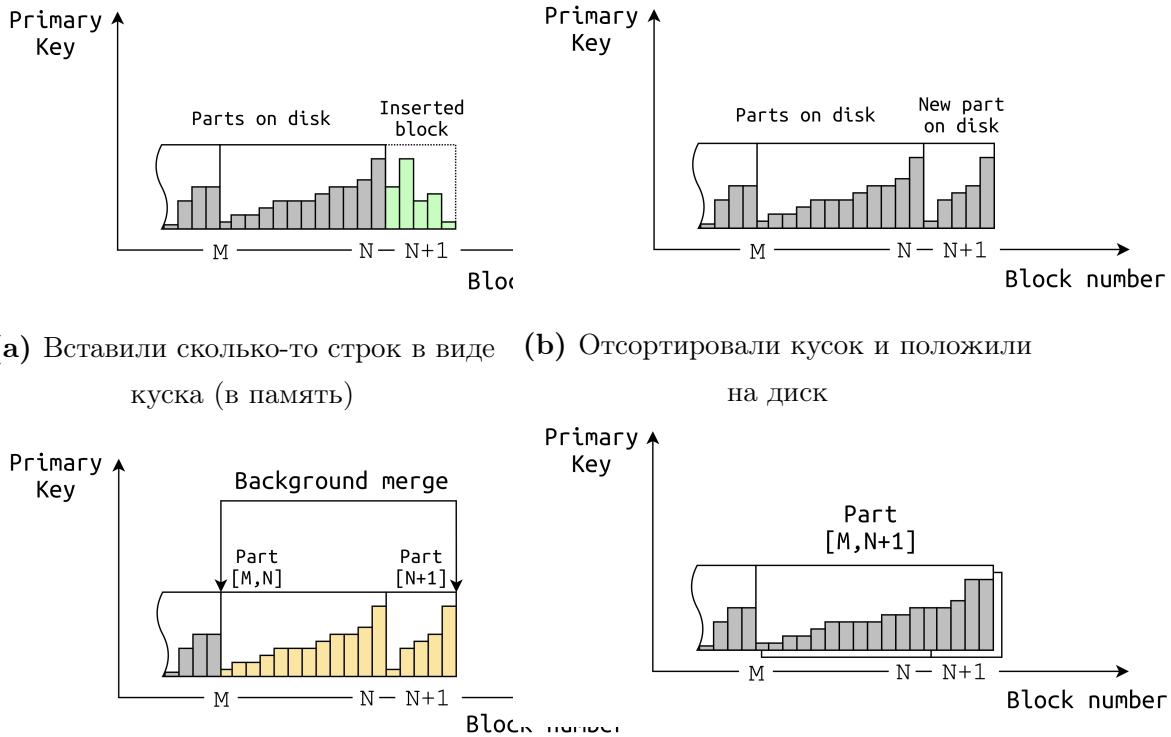


Figure 5: Пример RAID 6 над пятью дисками

3 Обзор архитектуры MergeTree в ClickHouse

MergeTree — семейство движков таблиц в ClickHouse, основанное на структуре хранения данных LSM Tree (log-structured merge tree). Основная идея этого движка таблиц заключается в том, что данные поступают большими *кусками (parts)* — по много строк сразу. Кусок сортируется по ORDER BY выражению таблицы (оно является обязательным параметром при создании MergeTree таблицы) и кладётся на диск. После множества записей таких кусков отсортированные куски сливаются в новый кусок, включающий все существующие. На рис. 6 представлен процесс слияния нового куска с существующим.



(a) Вставили сколько-то строк в виде куска (в память)

(b) Отсортировали кусок и положили на диск

(c) Процесс фонового слияния

(d) После слияния получился новый кусок

Figure 6: Процесс слияния кусков в MergeTree

3.1 Архитектура хранения данных MergeTree

В ClickHouse данные таблицы делятся на *партиции* по значению выражения `expr` из параметра `PARTITION BY expr`. Например, по месяцу записи. Партиция представляет из себя набор *кусков (parts)*, в идеале один. В фоне происходит процесс слияния кусков внутри каждой партиции, поэтому когда все клиенты перестают писать в партицию, то рано или поздно все куски в ней сольются в минимальное допустимое количество (максимальный размер куска регулируется настройкой).

Внутри одного куска колонки хранятся отдельными файлами, которые состоят из последовательно записанных значений данных этой колонки в соответствии с порядком, установленным `ORDER BY` настройкой таблицы.

Колонки сжаты блоками с помощью комплексного алгоритма сжатия, который можно воспроизвести с помощью поставляемой утилиты `clickhouse-compressor`; блоки сжатия никакого отношения к данным внутри не имеют, просто являются единицей сжатия. Горизонтально кусок поделён на *гранулы* — набор последовательно идущих строк внутри одного куска. Для каждой колонки рядом хранится файл с засечками. В таком файле для каждой гранулы записана пара чисел: смещение начала сжатого блока в файле колонки и смещение начала гранулы в разжатом блоке.

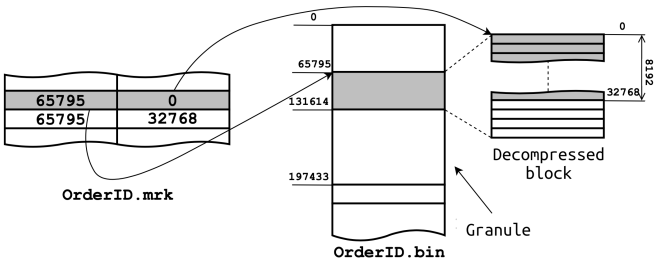
Для каждого куска хранится один файл индекса — отсортированные выражения `expr` из настройки `ORDER BY expr` для первой строки каждой гранулы. Таким образом, индекс сильно разреженный и хранит небольшое количество значений.

Для того, чтобы найти какую-то строчку по её PRIMARY KEY (который либо совпадает с ORDER BY, либо является его префиксом) [1], сначала вычисляется партиция, потом в каждом куске партиции с помощью бинарного поиска по индексу ищется гранула, в которой может находиться указанный PRIMARY KEY. После того, как становится известен список гранул-кандидатов, для каждой из них в засечках колонок ищутся смещения в файлах колонок, после чего в файле колонки можно быстро найти необходимые значения. Этот механизм представлен на рис. 7

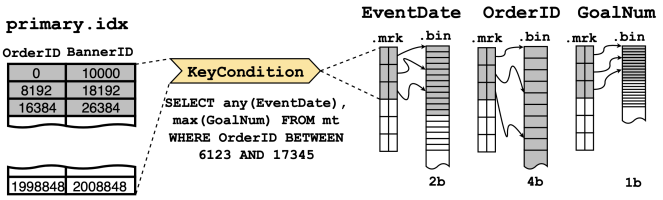
3.2 Реализация механизма хранения данных MergeTree

3.2.1 Движки таблиц

В ClickHouse существует множество различных движков таблиц, все они реализуют интерфейс `IStorage`. Часть движка MergeTree реализована с помощью класса `MergeTreeData`. Однако, в нём не имплементирована часть методов для работы с записью и чтением, так как таблицы MergeTree бывают обычные и реплицируемые. Это разделение обеспечивается двумя наследниками `MergeTreeData`: `StorageMergeTree` и



(a) Пример использования файлов засечек



(b) Механизм адресации с помощью индекса

Figure 7: Чтение засечек и адресация

`StorageReplicatedMergeTree` (рис. 8).

3.2.2 Потоки блоков

Транспорт данных в ClickHouse организован с помощью *блоков* — фиксированного набора строк одной таблички, который свободно помещается в оперативную память сервера. Собственно, блоки организованы в *потоки блоков* [2]. Потоки делятся на два типа: Output и Input и используют соответствующие базовые классы для реализации — `IBlockOutputStream` и `IBlockInputStream`.

В MergeTree и ReplicatedMergeTree для записи данных используются потоки `MergeTreeBlockOutputStream` и `ReplicatedMergeTreeBlockOutputStream` соответственно. Чтение происходит несколько иным способом, на нём остановимся позже.

3.2.3 Политики хранения и файловая система

Больше одного диска поддерживает только MergeTree. Для пользователя сервером ClickHouse, диск представляет из себя какой-то путь в файловой

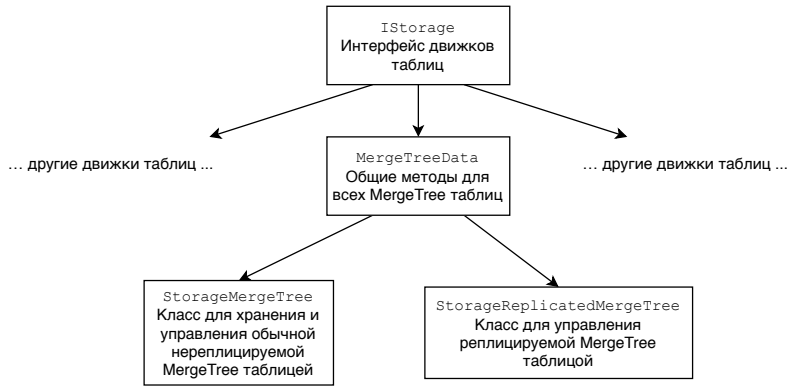


Figure 8: Классы для хранения и управления таблицами MergeTree

системе [3]. С недавних пор диск является абстракцией более широкого профиля — интерфейс `IDisk` является универсальной файловой системой, на базе которой реализована поддержка обычной работы с диском и работа с S3 хранилищами не только в качестве источника данных, но и в качестве основного хранилища табличных данных MergeTree.

Помимо `IDisk` и его реализаций в непосредственной работе с дисками принимает участие ещё две абстракции — `Volume` и `StoragePolicy`.

`Volume` является простой абстракцией над множеством дисков. Единственной отличительной особенностью `Volume` от вектора дисков является то, что метод `Volume::reserve` резервирует место на дисках с помощью подхода Round-Robin — то есть выбирается следующий за последним использованным диском и начиная с него производятся попытки зарезервировать место. Если ни на одном диске до конца массива не удалось зарезервировать место, то резервирование не произошло.

`StoragePolicy` является абстракцией над несколькими `Volume`. Политики хранения по существу позволяют только зарезервировать место на одном из `Volume` политики, а так же хранить и получать т.н. `move_factor` — процент заполнения дисков, входящих в эту политику.

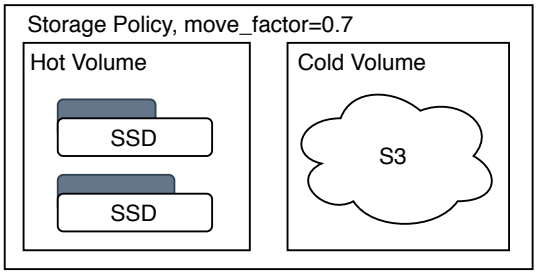


Figure 9: Пример политики с двумя `Volume`

3.2.4 Конфигурация дисков, вольюмов и политик

3.2.5 Запись данных в MergeTree

Запись блока на уровне партиций

Запись данных в нереплицируемой таблице MergeTree начинается с метода `MergeTreeBlockOutputStream::write`, в котором блок делится на блоки по партициям и они записываются на диск, для каждой партиции это новый кусок, который будет когда-то помёржен с остальными кусками партиции, поэтому по окончанию записи асинхронно вызывается поток, который занимается слиянием кусков в партициях.

Куски записываются с помощью метода `MergeTreeDataWriter::writeTempPart`. В нём получаются различные мета-данные, необходимые для записи, после чего происходит резервирование места на каком-то диске, выбор диска для записи куска выбирается с помощью Round-Robin (на самом деле это происходит внутри метода `Volume::reserve`, о котором мы уже писали ранее в секции про работу с дисками). После резервирования места на выбранном диске инициализируется кусок с помощью метода `MergeTreeData::createPart`, то есть создаются все необходимые файлы и делаются дополнительные приготовления (например, создаются буферы для записи в файлы колонок, засечек, хэшсумм и т.п.). Запись непосредственно данных блока осуществляется с помощью метода `MergedBlockOutputStream::writeWithPermutation`.

Разные представления куска

Существует два метода представления куска: оригинальный и самый

частоиспользуемый `Wide` и редкоиспользуемый `Compact`. Отличие второго заключается в том, что вместо стандартного подхода с индивидуальным хранением колонок в отдельных файлах, `Compact` хранит все данные всех колонок в одном файле `data.bin`, равно как и все засечки для этих колонок в файле `data.mrk3`. Этот формат подходит только для хранения совсем небольших кусков до 10Мб, поэтому используется очень редко. Пока что не будем рассматривать поддержку этого формата, но и не отрицаем, что, например, RAID 1, можно будет дёшево реализовать для данного типа кусков.

Запись Wide кусков

Метод `MergeTreeData::createPart` инициализирует кусок нужного типа. После этого в конструкторе `MergedBlockOutputStream` из объекта куска с помощью метода `IMergeTreeDataPart::getWriter` выбирается правильный writer — в нашем случае `MergeTreeDataPartWriterWide`. Внутри конструктора для каждой колонки создаётся объект структуры `IMergeTreeDataPartWriter::Stream` — некий контейнер для набора буферов для записи в файл колонки и в файл засечек. В буферы внутри этой структуры внутри метода `MergedBlockOutputStream::writeWithPermutation` и происходит запись.

Если углубиться дальше, запись блока происходит в методе `MergeTreeDataPartWriterWide::write`. Она осуществляется поколоночно с помощью метода `MergeTreeDataPartWriterWide::writeColumn`. Как мы уже знаем, куски делятся на гранулы, поэтому на этом уровне запись осуществляется по гранулам, в соответствии с вычисленной их длиной в самом начале метода `MergeTreeDataPartWriterWide::write`. Данные гранулы записываются с помощью `IDataType::serializeBinaryBulkWithMultipleStreams`, который реализован индивидуально для нескольких типов данных. Внутрь этого метода передаётся структура настроек, содержащая функцию `getter`, отдающая необходимый `WriteBuffer`. Эта абстракция уже находится на уровне одного диска и с ней нам работать придётся только в качестве пользователя.

Для наглядности изобразим вызовы, происходящие в процессе записи на диаграмме (рис. 10):

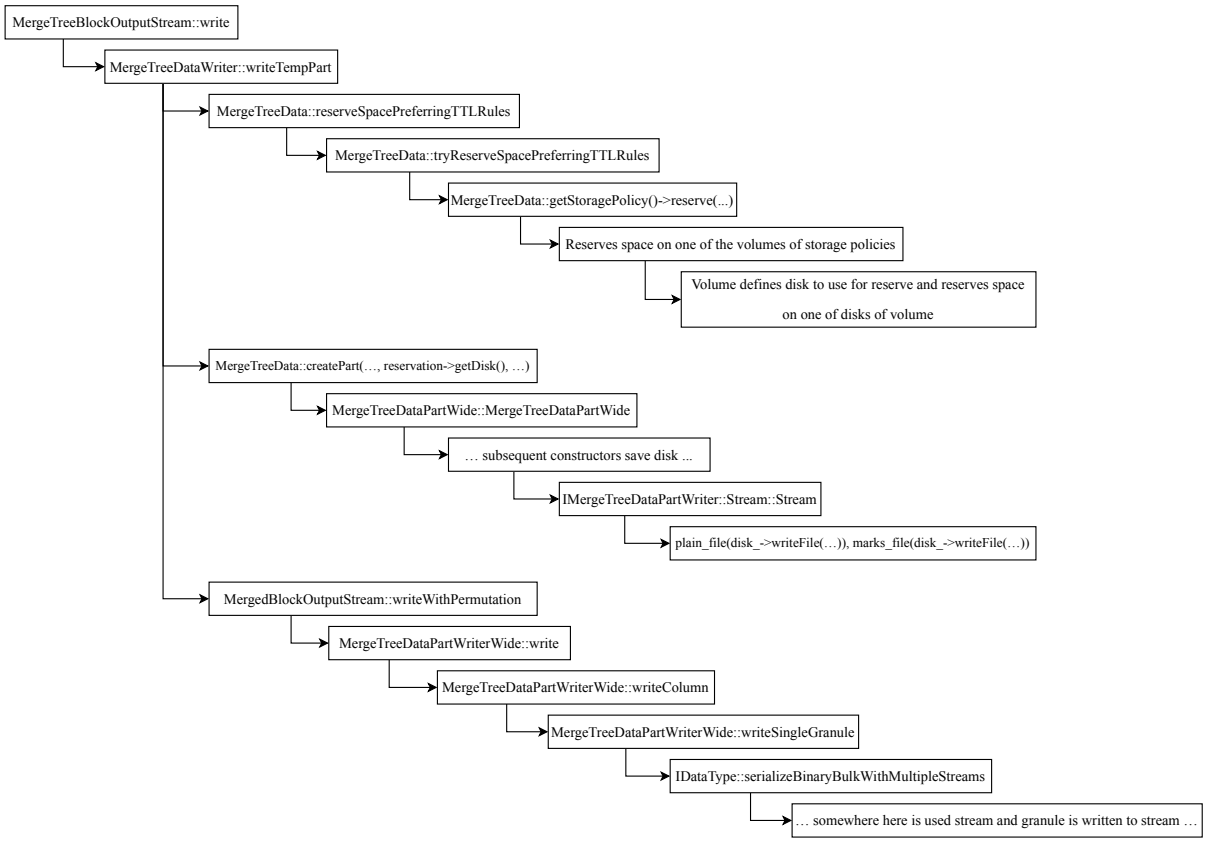


Figure 10: Дерево вызовов функций

3.2.6 Чтение данных в MergeTree

Процессоры Это свежая концепция, которая создана для того, чтобы ускорить процесс чтения данных с диска. Раньше использовались потоки блоков, с которыми мы уже раньше встречались в процессе записи. Там идея была в том, что потоки легко надстраивались друг над другом, создавая конвейер записи / чтения с помощью потока. Эта идея легла в основу *процессоров и пайплайнов процессоров* (см. `IProcessor` и `QueryPipeline`.

Процессор это исполняющая единица, принимающая на вход вектор

InputPorts и пишущая результат исполнения в OutputPorts. Порты в данном случае это очень простая абстракция, взятая из ОС — к ним можно подключаться, писать (OutputPort::pushData) или скачивать (InputPort::pullData) данные. Соответственно, абстрактный процессор имеет множество портов на вход и множество на выход. Фактически, поверх этой абстракции можно реализовывать совершенно разные процессоры. Например, абстрактный источник данных это процессор без входящих портов и с одним выходящим, или фильтрующий процессор, имеющий какое-то условие фильтрации данных, один входящий и один выходящий порт. Большее количество примеров можно прочитать в документации к классу IProcessor прямо в заголовочном файле src/Processors/IProcessor.h.

Pipes, QueryPipeline и PipelineExecutor

Pipe это набор процессоров, имеющий один выходной порт, все внутренние порты процессоров соединены друг с другом, представляя некий аналог обычной трубы. Например, Pipe может состоять из одного процессора — процессора чтения куска с диска.

QueryPipeline это управляющая обёртка над множеством пайпов. Исполнение пайплайна происходит асинхронно — QueryPipeline::execute возвращает объект класса PipelineExecutor, который уже имеет метод execute, его вызов синхронно выполнит пайплайн. Под PipelineExecutor::execute скрывается фиксированный набор потоков, для которых пайплайн разбивается на чанки обработки и все они ставятся в очередь. Это позволяет набору потоков равномерно распределить нагрузку между собой, тем самым утилизируя доступный процессор максимально.

Чтение куска

Будем разбирать процесс чтения на примере SELECT запроса.

Процесс чтения данных асинхронный — сначала строится пайплайн чтения, учитывая все входные данные (пользовательский запрос, настройки движков и т.д.), после чего этот пайплайн выполняется с помощью вышеупомянутого PipelineExecutor.

Весь конвеер исполнения чтения несколько сложнее, чем для записи 11.

Это во многом обусловлено дополнительными абстракциями для ускорения чтения. В верхней части по большому счёту строится базовый пайплайн, состоящий из `MergeTreeSelectProcessor`. Ниже уже во время исполнения пайплайна, внутри процессора генерируется задача на чтение куска и чтение происходит в `MergeTreeBaseSelectProcessor::readFromPart`. С этого момента процесс чтения становится похож на процесс записи кусков, описанный выше, общие детали можно заметить на диаграммах.

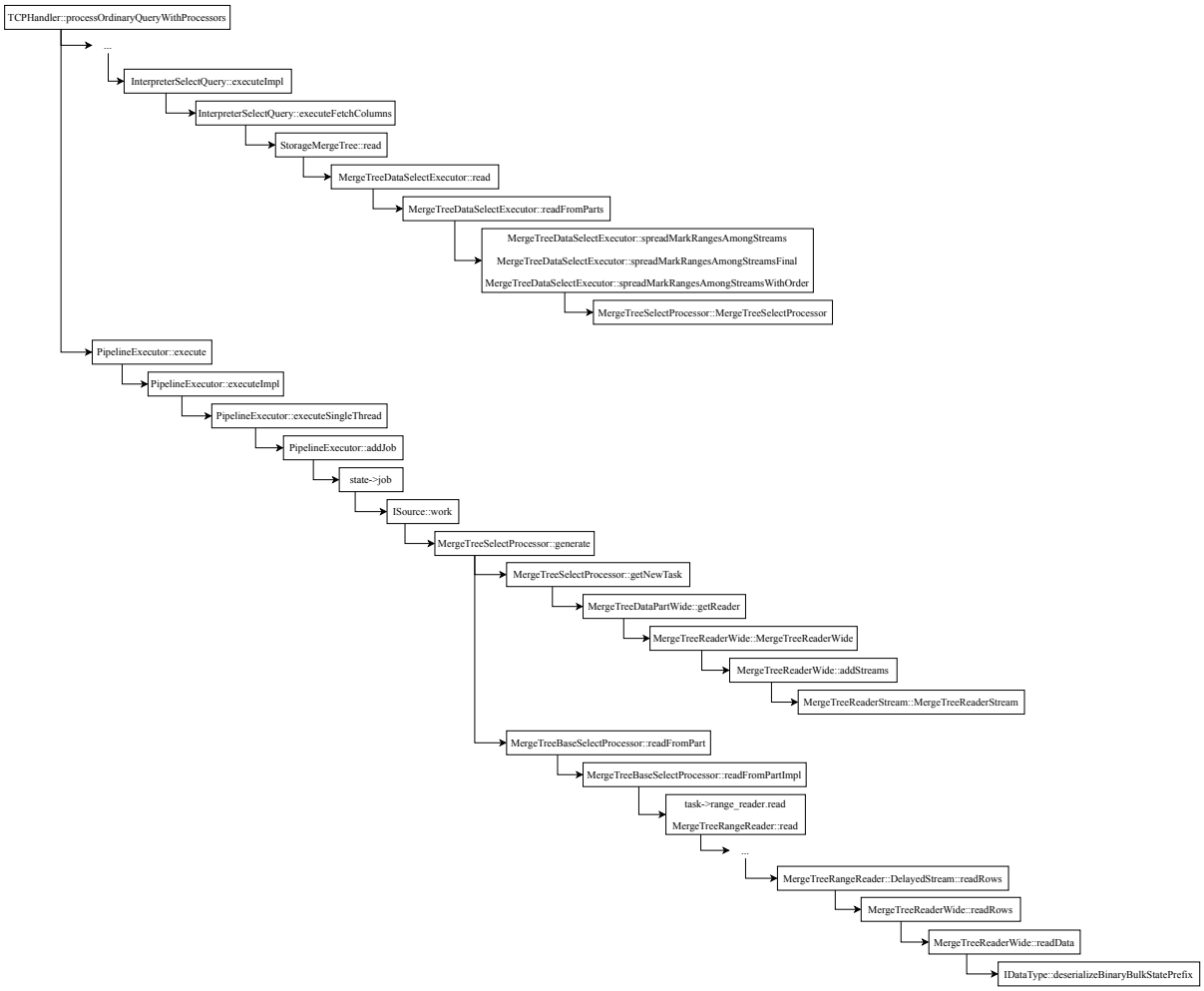


Figure 11: Дерево вызовов функций

3.2.7 Слияние и перемещение данных в MergeTree

Слияние кусков

Как уже было описано ранее, слияние это один из самых важных процессов, происходящих в MergeTree. Оно необходимо для поддержания небольшого количества отсортированных кусков.

Слияние множества кусков подразумевает чтение из множества кусков и запись одного большого, также отсортированного целиком. Точкой входа в слияние можно считать либо `OPTIMIZE` запрос, либо поток, который просыпается специально для осуществления операции слияния. Ожидаемо, что и там и там используется один и тот же механизм, поэтому рассмотрим процесс слияния на примере `OPTIMIZE` запроса.

Точкой входя для `OPTIMIZE` запросов является `StorageMergeTree::optimize`. По большому счёту, там происходят какие-то приготовления и непосредственное слияние с помощью `MergeTreeDataMergerMutator`. В этом классе содержатся методы, позволяющие собрать список кусков, подходящих для слияния (как минимум, эти куски должны быть в одной партиции, а так же удовлетворять различным настройкам движка), а так же произвести процесс слияния с помощью метода `mergePartsToTemporaryPart`. В этом методе можно пронаблюдать примеры использования чтения и записи в одном месте.

Перемещение кусков

Перемещение кусков это дополнительная функциональность, описанная ранее в секции про диски и вольюмы. Она заключается в возможности перемещения кусков по истечению какого-то срока или степени заполнения диска на другой диск, например, с большим объёмом в угоду меньшей скорости.

Этот процесс осуществляется с помощью `MergeTreePartsMover` и на самом деле является простыми операциями с файловой системой, по крайней мере когда данные куска лежат на одном диске. В случае, когда данные куска будут лежать более, чем на одном диске, появится необходимость реализовать аналогичный слиянию механизм, но только с одним куском

— то есть прочитать кусок и записать его на другой вольюм.

4 Реализация RAID в коде ClickHouse

Реализация классического RAID в ClickHouse не очень интересна, так как это реально сделать с помощью, например, `mdraid`. Интересно реализовать его, зная про внутреннее устройство MergeTree, а именно — про гранулы. Большие файлы кусков можно бить на блоки не по байтам, а по гранулам. Это в теории позволит достаточно просто распараллелить процесс записи и чтения, а в рамках чтения / записи одной гранулы не придётся переключаться между дисками.

Такимо образом, при использовании, например, RAID 0, один кусок будет присутствовать на нескольких дисках, а в качестве блоков записи будут использоваться гранулы.

Чтобы реализовать абстракцию, позволяющую поддержать как существующую политику для множества дисков (round-robin размещение кусков), так и RAID-like политики, необходимо будет затронуть многие описанные выше места.

4.1 Этапы реализации

Поскольку задача довольно большая, видятся следующие этапы реализации поддержки RAID:

1. Придумать подходящие абстракции для процесса записи
2. Придумать подходящие абстракции для процесса чтения
3. Реализовать и встроить новые абстракции в процесс записи, сохранив текущее поведение, то есть с одним возможным режимом работы (запись кусков через round-robin), но подразумевая появление новых вариантов поведения с возможностью их конфигурации.
4. Аналогичный предыдущему пункт про процесс чтения.

5. Реализовать RAID в порядке RAID 1, RAID 0, RAID 5.

4.2 Обновление `Volume`

В первую очередь необходимо обновить класс `Volume`. Как уже было описано ранее, функциональность `Volume` сводится к хранению конфигурации вольюма и резервированию места на нужном диске. Здесь можно завести интерфейс `IVolume` и на его базе реализовать первый `VolumeJBOD`. Содержательное отличие реализаций интерфейса `IVolume` будет заключаться в различной реализации метода `IVolume::reserve`. Например, в `VolumeRAID1::reserve` место будет резервироваться сразу на нескольких дисках в одинаковом количестве.

Зарезервированное место передаётся с помощью реализаций интерфейса `IReservation`, для каждого типа дисков он реализуется по-своему. Чтобы корректно передавать зарезервированное место, потребовалось модифицировать `IReservation` — теперь внешний интерфейс выглядит таким образом, что в `IReservation` может находиться больше одного диска, были добавлены методы `getDisks` и `getDisk(size_t i)`.

4.3 Обновление конфигурации

Чтение конфигурации происходит довольно просто — есть объект класса `Poco::Util::AbstractConfiguration`, хранящий в себе конфигурацию, прочтённую из XML. Секции в нём адресуются с помощью строки, состоящей из ключей секции латиницей, разделёнными через знак точки. Для поддержки конфигурации RAID достаточно было реализовать метод `createVolumeFromConfig` в `src/Disks/createVolume.h`, который на основе конфигурации создаёт объект нужного класса (наследника `IVolume`).

4.4 Модификация процесса записи в MergeTree

Объект диска хранится и используется во многих местах, начиная с `IMergeTreeDataPart`, поэтому в первую очередь необходимо заменить его

на объект вольюма, который хранит в себе несколько дисков. Этот вольюм будет браться не из настроек, а конструироваться для каждого объекта `IReservation`, так как резервирование места происходит до создания куска. Таким образом, в объекте куска будет храниться актуальный вольюм, на котором есть место для записи нового куска в том виде, который указан в конфигурации (JBOD или RAID 0, 1, 5, 6). Реализуем так же один диск в виде вольюма `SingleDiskVolume`, который будет использоваться в классическом режиме работы над одним диском.

За создание файлов колонок на дисках отвечает `IMergeTreeDataPartWriter::Stream`, а непосредственно в эти файлы инициирует запись реализация интейрфейса `IMergeTreeDataPartWriter`. Поскольку сейчас весь кусок пишется на один диск, то `Stream` тоже создаёт файлы и сохраняет буферы для одного диска. В текущей реализации записи интерфейс `IMergeTreeDataPartWriter`, его реализации (`*Wide` и `*Compact`) и `IMergeTreeDataPartWriter::Stream` очень сильно связаны, поэтому можно пойти по пути наименьшего сопротивления — унаследоваться от `MergeTreeDataPartWriterWide` и переопределить в нём процесс записи колонки, добавив туда выбор диска для гранулы, изменить механизм вычисления засечек и индексов. Таким образом мы выделили класс `MergeTreeDataPartWriterWideSingleDisk`, который реализует текущее поведение (с JBOD) и начали реализацию классов `MergeTreeDataPartWriterWideRA`
которые работают с множеством дисков в вольюме.

4.5 Модификация процесса чтения в MergeTree

Новый процесс записи реализован таким образом, что без знания о том, что файлы колонок, засечек и индекса созданы с помощью RAID, можно подумать, что эти файлы представляют просто отдельные куски. Процесс чтения уже реализован достаточно умно и быстро, чтобы добавлять туда новый параллелизм. Ну действительно, давайте вместо этого встроимся в процесс чтения после проверки чексумм кусков и просто представим, что разложенные по разным дискам части одного куска это просто разные куски (исключительно для реализации быстрого чтения).

Чтение кусков начинается с `MergeTreeDataSelectExecutor::readFromParts`. Этот метод очень большой и плохо декомпозированный, к сожалению. Там берутся реальные куски на диске и нарезаются на промежутки по гранулам таким образом, чтобы пул потоков мог эффективно читать эти промежутки. Нарезка кусков на промежутки происходит после проверки чексумм, поэтому достаточно будет просто поддержать нарезку на промежутки в новой парадигме — когда файлы колонок лежат на разных дисках. Этой модификации будет достаточно для того, чтобы обеспечить быстрое чтение с множества дисков.

5 Дальнейшие разработки

5.1 Другие типы RAID

На базе существующего решения можно реализовать поддержку практических любых типов RAID, даже с использованием дополнительных контроллеров, при условии наличия C/C++ API у этих контроллеров. Для реализации новой политики записи на вольюм с несколькими дисками достаточно реализовать ещё один класс, унаследовавшись от `MergeTreeDataPartWriterWide` и имплементировав методы `writeColumn`, `writeSingleGranule` и другие (см. в реализации `MergeTreeDataPartWriterWideSingleDisk` или `MergeTreeDataPartWriterW`

5.2 Восстановление реплик

Одной из ключевых возможностей движка MergeTree является возможность настроить распределённое хранилище данных, с использованием кластера Zookeeper. Процедура восстановления повреждённой реплики распределённой таблицы заключается в полном скачивании данных с других реплик (соответственно, с перезаписью существующих данных, если хоть какие-то остались). При выходе из строя одного из дисков, можно будет не инвалидировать все данные на реплике, а восстанавливать данные с помощью стандартной процедуры локального восстановления диска в выбранном типе RAID.

Это позволит не качать данные по сети, тем самым сэкономив трафик и ускорив процедуру восстановления, так как чтение с диска на порядки быстрее чтения данных по сети.

6 Заключение

На первый взгляд задача, которая стояла перед нами, была небольших объёмов и ограничивалась изменением процедур записи и чтения данных MergeTree с дисков. К сожалению, после погружения и подробного исследования механизмов в MergeTree, это оказалось не совсем так. Нами была проделана большая работа по исследованию механизмов работы с данными в MergeTree, а так же по подготовке этих механизмов к работе с несколькими дисками. Более того, были подготовлены абстракции для реализации записи и чтения с использованием нескольких дисков и реализован прототип RAID 1.

References

[1] *MergeTree: Choosing a Primary Key That Differs From the Sorting Key*, ClickHouse Documentation, `https://clickhouse.tech/docs/en/engines/table_engines/mergetree_family/mergetree/#choosing-a-primary-key-that-differs-from-the-sorting-key`

[2] *Overview of ClickHouse Architecture, Block Streams*, ClickHouse Documentation, `https://clickhouse.tech/docs/en/development/architecture/#block-streams`

[3] *MergeTree, Using Multiple Block Devices For Data Storage*, ClickHouse Documentation, `https://clickhouse.tech/docs/en/engines/table_engines/mergetree_family/mergetree/#table_engine-mergetree-multiple-volumes`

[4] David A. Patterson, Garth Gibson and Randy H. Katz, *A Case for Redundant Arrays of Inexpensive Disks (RAID)*, Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data 109-116, 1988.