**Москва 2020**

**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

# Оптимизация параллельного GROUP BY с помощью flat combining

**Выполнил студент группы БПМИ163, 4 курса,**

**Серебряков Максим Викторович**

**Руководитель ВКР:**

**приглашенный преподаватель, Миловидов Алексей Николаевич**

# Оглавление

*Введение* 3

*1 Обзор существующих подходов* 5

1.1 Тривиальный метод 5

1.2 Тривиальный параллельный метод 6

1.3 Two-level метод 6

*2 Реализованные новые подходы* 8

2.1 Splitting-aggregator 8

2.2 Splitting-aggregator с быстрым делением 8

2.3 Локальные хеш-таблицы + общая Two-level хеш-таблица 9

2.4 Локальные хеш-таблицы + Splitting aggregator с очередями 10

2.5 Локальные хеш-таблицы с буфферами + общая Two-level хеш-таблица на основе Flat Combining 11

2.5.1 Flat Combining 11

2.5.2 Локальные хеш-таблицы с буфферами + общая Two-level хеш-таблица 12

*3 Сравнение реализованных подходов* 14

3.1 Описание способа тестирования 14

3.2 Результаты сравнения для набора данных WatchID 15

3.3 Результаты сравнения для набора данных RefererHash 17

3.4 Результаты сравнения для набора данных SearchPhrase 18

3.5 Выводы 19

*4 Реализация и применение в ClickHouse* 20

4.1 Устройство агрегации в ClickHouse 20

4.2 Особенности применения нового метода 21

4.3 Тестирование 23

4.4 Выводы 24

*5 Заключение* 25

*6 Источники* 27

# Введение

ClickHouse – это столбцовая система управления базами данных (СУБД) для онлайн обработки аналитических запросов (OLAP), разработанная в компании Яндекс [1]. Одним из самых базовых и популярных типов запросов является агрегация данных по ключу – данные делятся на группы с одинаковыми значениями ключей, а затем для каждой группы вычисляется агрегатная функция, например, количество элементов в группе, сумма значений по какому-либо полю среди элементов группы и так далее. В ClickHouse ключи, по которым нужно сделать агрегацию, указываются в SQL-запросе в секции GROUP BY. Разумеется, такая важная функциональность должна работать быстро и эффективно на больших объемах данных. Это довольно сложная и очень интересная задача, ведь не существует единственного верного решения, которое будет работать эффективнее других на всех возможных наборах данных. Ускорение работы GROUP BY будет полезно для всех пользователей ClickHouse, так как позволит производить расчеты с агрегацией еще быстрее.

Цель работы – попытаться ускорить текущий подход любыми новыми способами, возможно только для определенных случаев.

Для достижения цели в рамках данной работы были поставлены следующие задачи:

- Реализация различных подходов для ускорения группировки данных в запросах с использованием GROUP BY
- Исследование метода flat combining на возможное его применение в ClickHouse
- Проведение сравнительного анализа реализованных подходов
- Реализация и применение самого эффективного алгоритма в ClickHouse в формате созданного pull-request в репозиторий ClickHouse на github.com

- Проверка реализованного метода на функциональных тестах и тестах на производительность

Работа структурирована следующим образом:

В первой главе описываются базовые алгоритмы параллельной агрегации и алгоритм, который используется в ClickHouse сейчас.

Во второй главе описываются новые подходы к параллельной агрегации, их особенности и области применения.

В третьей главе описывается методология тестирования алгоритмов, предложенных во второй главе, приводятся результаты сравнительного анализа на различных наборах данных.

В четвертой главе описывается детали реализации и применения лучшего алгоритма по эффективности из предложенных непосредственно в самом ClickHouse.

Ключевые слова – многопоточность, хеш-таблица, агрегация, параллельные вычисления, оптимизация

# 1 Обзор существующих подходов

Существует большое количество различных алгоритмов, структур данных и подходов, которые с тем или иным успехом решают задачу агрегации набора данных по ключу. Выбор метода обычно определяется скоростью работы, объемом используемой памяти и для каких данных предполагается его использовать.

Самый базовый алгоритм заключается в следующем: кладем наши данные в обычный массив, затем сортируем его по ключу агрегации, а после этого проходим вдоль него, попутно считая агрегатную функцию, так как элементы с одинаковыми ключами лежат подряд. Достоинством данного алгоритма является его простота, но он обладает большими недостатками: большое время работы $O(n \log n)$ и всегда тратится $O(n)$ памяти, где $n$ – количество элементов, даже когда уникальных значений ключей очень мало.

Для данной задачи есть более подходящая структура данных, чем обычный массив – это ассоциативный массив. Это особый тип данных, позволяющий хранить пары вида “(ключ, значение)”, а так же искать, добавлять и удалять их. У ассоциативного массива существует много реализаций, самые популярные из них: lookup-таблица, хеш-таблица, бинарное дерево, список с пропусками (skip list), Би-дерево. Но снова разные реализации подходят для разных случаев. Для того, чтобы сделать агрегацию, нужны только две операции – быстрый поиск и добавление, и что самое главное, абсолютно не важен порядок элементов. Поэтому самым подходящим вариантом является хеш-таблица со скоростью этих двух операций $O(1)$ и используемой памятью $O(m)$, где $m$ - количество уникальных ключей.

## 1.1 Тривиальный метод

Тривиальный способ использования хеш-таблицы заключается в следующем: изначально у нас пустая хеш-таблица, перебираем по очереди элементы исходных данных, ищем очередной ключ в хеш-таблице, если не

нашли, то добавляем его в хеш-таблицу со значением агрегатной функции от одного элемента, если нашли, то обновляем значение агрегатной функции. Данный способ уже имеет большие преимущества по сравнению с сортировкой: скорость работы $O(n)$, где $n$ – количество элементов, объем потребляемой памяти $O(m)$, где $m$ – количество уникальных ключей. Но он имеет существенный недостаток – ресурсы процессора используются неэффективно, все вычисления происходят в одном потоке, поэтому при большом объеме данных данный алгоритм работает долго.

## 1.2 Тривиальный параллельный метод

Для того, чтобы получить ускорение и меньше зависеть от объема исходных данных, вычисления следует производить параллельно в нескольких потоках. Самым простым и прямо следуемым из определения параллельности алгоритмом является разбиение исходных данных на части, параллельное вычисление для каждой из них, а затем объединение результатов в один. Более строго говоря, каждый поток независимо работает со своим куском данных, агрегирует их, используя свою собственную хеш-таблицу, а затем в одном потоке все эти хеш-таблицы объединяются в одну. Данный способ прост в реализации, так как не надо заботиться о потоковой безопасности (thread-safety), прибегая к различным способам синхронизации, что является большим плюсом. Узким местом данного алгоритма является стадия объединения, так как она происходит в одном потоке последовательно. Если доля уникальных ключей велика (например, больше 30%), то она и вовсе занимает время как в тривиальном способе, при этом еще и потратив время на стадию параллельной агрегации. Но если же доля мала, то алгоритм работает очень быстро.

## 1.3 Two-level метод

В СУБД ClickHouse реализован метод, который обходит проблему долгого объедения хеш-таблиц в одну, при этом сохраняя скорость

параллельной агрегации и обходясь без способов синхронизации потоков. В основе его лежит так называемая Two-level хеш-таблица – это хеш-таблица, которая состоит из 256 обычных хеш-таблиц, что и объясняет ее название. Добавление элемента в данную хеш-таблицу происходит следующим образом: сначала вычисляется хеш-значение – номер дочерней хеш-таблицы (от 0 до 255), а затем ключ вставляется в хеш-таблицу под этим номером (с использованием своей хеш-функции). Благодаря этому, достигается очень важное свойство – каждый ключ принадлежит только своей дочерней хеш-таблице, в другую он попасть не может. Поиск происходит аналогично: вычисляется номер дочерней хеш-таблицы, а затем в ней ищется элемент. Сам алгоритм заключается в следующем:

1) Стадия подготовки: исходные данные делятся на части (например, ровно на $k$ частей, где $k$ – количество потоков). Создаются $k$ Two-level хеш-таблиц. Данная стадия не занимает времени.
2) Стадия агрегации: каждый поток агрегирует свою часть, используя свою Two-level хеш-таблицу. На выходе получаются $k$ Two-level хеш-таблиц ($k * 256$ обычных внутри них).
3) Стадия объединения: каждому потоку назначается номер дочерней хеш-таблицы, и он объединяет $k$ дочерних хеш-таблиц с этим номером в одну. То есть 256 номеров добавляются в очередь, и $k$ потоков их разбирают параллельно. Независимость объединения от дочерних хеш-таблиц с другими номерами достигается за счет того, что в каждой из дочерних хеш-таблиц с одинаковым номером лежат ключи из одной группы, которых нет в других хеш-таблицах по свойству выше. В итоге получаем одну итоговую Two-level хеш-таблицу.

Данный метод очень эффективен, так как и стадия агрегации и стадия объединения выполняются параллельно без синхронизации потоков. Но для

малого количества данных нет смысла его использовать, поэтому в ClickHouse сначала используется тривиальный способ, а после определенного порога по количеству ключей и по использованию памяти  (которые пользователь может поменять, если захочет) хеш-таблица конвертируется в Two-level и используется данный алгоритм.

# 2 Реализованные новые подходы

В рамках данной работы были реализованы несколько новых подходов.

## 2.1 Splitting-aggregator

Алгоритм можно описать так:

1) Стадия подготовки: сначала для всех ключей считаются значения хеш-функций, находится минимальное и максимально значение, и ключи разбиваются на $k$ групп по значению хеш-функции (где $k$ – количество потоков).
2) Стадия агрегации: каждый поток параллельно с другими потоками обрабатывает только ключи из своей группы, агрегирует их в свою собственную хеш-таблицу. На выходе получаются $k$ хеш-таблиц.
3) Стадия объединения: не производится, так как в разных хеш-таблицах находятся разные ключи (ни у одной пары хеш-таблиц нет хотя бы одного общего ключа).

Главными достоинствами данного алгоритма являются простота реализации и отсутствие стадии объединения. Но он обладает существенным недостатком: если доля уникальных ключей мала (например, меньше 10%), то одному потоку достанется группа, в которой будет гораздо больше элементов, чем в остальных, т.е. большинство потоков завершатся мгновенно, а один будет агрегировать свою большую часть данных. Когда же различных ключей много, то данный алгоритм работает очень эффективно.

## 2.2 Splitting-aggregator с быстрым делением

В “Splitting-aggregator” предварительный подсчет значений хеш-функции для всех ключей выполняется в одном потоке и при большом количестве данных может занимать ощутимое количество времени. Альтернативный подход к разбиению заключается в том, что номер потока, которому принадлежит ключ, определяется путем взятия остатка от деления значения хеш-функции на количество потоков. Но деление – дорогостоящая операция в

плане производительности. Поэтому, в качестве замены, была использована формула, которая так же возвращает число от 0 до $k$ - 1, с таким же распределением, как и взятие остатка от деления, но работающая в 4 раза быстрее: $x * k \gg 32$, где $x$ – значение хеш-функции, $k$ – количество потоков, $\gg$ - операция битового сдвига вправо [1].

### 2.3 Локальные хеш-таблицы + общая Two-level хеш-таблица

Суть данного метода заключается в следующем:

1) Стадия подготовки: данные делятся на $k$ равных частей, где $k$ – количество потоков. Заводится одна общая Two-level хеш-таблица. Никаких вычислений не производится.
2) Стадия агрегации: каждый поток сначала добавляет в свою собственную локальную хеш-таблицу ключи, но делает это до момента, пока ее размер (количество уникальных ключей) меньше порога (например, 4096). Если же размер локальной хеш-таблицы больше, то тут два варианта:
   a. Если ключ имеется в локальной хеш-таблице, то он кладется туда (пересчитывается значение агрегатной функции)
   b. Если же в локальной хеш-таблице ключа нет, то производится попытка вставить его в общую Two-level хеш-таблицу, при этом взяв блокировку на определенную дочернюю хеш-таблицу, используя примитивы синхронизации потоков. Это необходимо, так как в одинаковый момент времени разные потоки могут вставлять ключ в одну и ту же дочернюю хеш-таблицу, что без блокировки приведет к аварийному завершению программы. Если же взять блокировку не удалось, то ключ вставляется в локальную хеш-таблицу.
3) Стадия объединения: по очереди объединяются все локальные хеш-таблицы с общей Two-level хеш-таблицей в одном потоке.

Локальные хеш-таблицы помогают справиться с часто встречаемыми ключами. Если ключ часто встречается в исходных данных, то с большой вероятностью он встретится до достижения порога и попадет в локальную хеш-таблицу, а значит большая часть данных будет агрегироваться без синхронизации.

Стандартными способами синхронизации потоков являются мьютекс, семафор и условная переменная. Но в данном случае, блокировка должна быть быстрой и легковесной, но в то же время обеспечивающей последовательный доступ к общим ресурсам. Самым эффективным в данном случае способом является атомарная переменная, которую можно переводить из состояния “ресурс захвачен” в состояние “ресурс освобожден” и обратно, с помощью атомарной операции “сравнение с обменом” (CAS). Получается lock-free алгоритм.

Альтернативным подходом к стадии объединения является параллельное объединение в общую хеш-таблицу, но для этого потребуется использовать примитивы синхронизации. Так же предполагается, что локальные хеш-таблицы не должны быть очень большими, хотя после превышения порога в них периодически добавляются новые элементы. Главный минус данного алгоритма: чем больше доля уникальных ключей, тем медленнее он работает, так как потоки будут сильно конкурировать, а локальные хеш-таблицы станут большими, что приведет к замедлению стадии объединения.

### 2.4 Локальные хеш-таблицы + Splitting aggregator с очередями

Данный метод за основу берет один из Splitting aggregator алгоритмов, и пытается решить проблему, когда один поток обрабатывает слишком большую часть данных, при этом используя локальные хеш-таблицы и синхронизацию, как в предыдущем методе. Описать его можно так:

1) Стадия подготовки: данные делятся на $k$ групп, где $k$ – количество потоков. Создаются $k$ глобальных хеш-таблиц, каждая

предназначенная для своего потока, и содержащая только свои ключи. Создаются $k$ очередей для каждого потока, то есть всего $k * k$ очередей.

2) Стадия агрегации: работа с локальной хеш-таблицей происходит аналогично предыдущему методу, но с отличием, когда в локальной хеш-таблице уже больше порога ключей и очередной ключ в ней не нашелся. Для такого ключа определяется номер потока, которому он принадлежит: если текущему, то элемент кладется в глобальную хеш-таблицу для текущего потока, если же другому, то добавляется в очередь для этого потока, то есть в очередь с индексами $[i,j]$, где $i$ – номер текущего потока, а $j$ – номер потока, которому принадлежит данный ключ. Собрав какое-то количество ключей для других потоков, для диапазона с ними в буффере выставляется флажок (атомарная переменная), что означает, что диапазон готов для чтения. Потоки раз в какое-то количество итераций смотрят на готовые для них диапазоны в буффере, читают ключи из них и добавляют в свою часть глобальной хеш-таблицы.
3) Стадия объединения: в глобальных хеш-таблицах уже лежат разбитые на группы ключи, поэтому остается только добавить в них ключи из локальных хеш-таблиц. Так как локальные хеш-таблицы маленькие, то объединение происходит в одном потоке без синхронизации.

Данный метод интересен тем, что объединение в конце происходит почти мгновенно, нет настоящих блокировок (алгоритм lock-free), данные равномерно распределены между потоками. Но есть и минусы: реализация сложнее, чем у других алгоритмов, расходуется слишком много памяти, плохая кеш-локальность. В итоге минусы сильно обесценивают плюсы.

## 2.5 Локальные хеш-таблицы с буфферами + общая Two-level хеш-таблица на основе Flat Combining

### 2.5.1 Flat Combining

Flat combining – это новая парадигма синхронизации для построения конкурентных структур данных [3]. Основная идея данного подхода заключается в следующем: имеется мьютекс и список анонсов (односвязный список) размером, пропорциональным количеству потоков. Каждый поток при первом обращении к структуре данных, с которой он работает, создает запись в списке анонсов. Когда потоку надо произвести операцию над структурой данных, он размещает запрос в его записи, например для стека операции push (добавление элемента в конец) и pop (получение элемента с конца) с их аргументами, и пытается захватить мьютекс. Если мьютекс захвачен, то поток становится так называемым комбайнером, он проходит по всему списку анонсов, выполняет все запросы из него, записывая результат операции обратно в список анонсов, и в итоге освобождает мьютекс. Если же мьютекс захватить не удалось, то поток начинает ждать до тех пор, пока комбайнер не выполнит его запрос и не вернет результат. Запросы операций размещаются в существующей записи списка анонсов, которая находится в локальном хранилище потока (thread local storage). Иногда комбайнер удаляет из списка анонсов старые записи. Еще одной важной особенностью *flat combining* является поглощающая стратегия (elimination back-off), которая возможна благодаря комбайнеру: взаимоисключающие операции, например, push и pop для стека, можно реализовать без реального выполнения их на структуре данных.

Подводя итог, можно сказать, что с помощью *flat combining* можно сделать из обычной структуры данных эффективную lock-free структуру данных, в который будут только атомарные операции без примитивов синхронизации.

С помощью *flat combining* можно реализовать lock-free хеш-таблицу, но в данном случае она не даст существенно прироста производительности, так как результат операции не нужен (происходит только добавление элемента), нет слишком большой конкурентности (в большинстве случаев количество потоков не превышает 32), невозможно реализовать поглощающую стратегию как, например, для стека, да и в целом очень сложный код, на порядок сложнее других методов. Но идея с отложенными операциями и последующим массовым выполнением была использована для создания собственного алгоритма.

### 2.5.2 Локальные хеш-таблицы с буфферами + общая Two-level хеш-таблица

Данный метод является улучшением третьего метода ("Локальные хеш-таблицы + общая Two-level хеш-таблица") и старается сократить время работы стадии объединения, путем применения идеи, схожей с *flat combining*. Описать его можно так:

1) Стадия подготовки: такая же, как в третьем методе.
2) Стадия агрегация: аналогичная третьему методу, но с существенным изменением – если потоку не удается получить блокировку на дочернюю хеш-таблицу в Two-level хеш-таблице, то он кладет ключ не в локальную хеш-таблицу, а в свой локальный буффер для определенной дочерней хеш-таблицы. Если же удается взять блокировку, то он добавляет все ключи из буффера для этой же дочерней хеш-таблицы вместе с текущим ключом в нее за раз. Этим как раз метод и похож на *flat combining*, где операции выполняются комбайнером массово. После того, как все элементы обработаны, оставшиеся ключи из буфферов, которые не были добавлены в общую хеш-таблицу, добавляются в локальную.
3) Стадия объединения: аналогичная третьему методу.

Несмотря на схожесть методов, отличия в производительности существенные. Благодаря отличию в стадии агрегации, локальные хеш-таблицы не разрастаются и почти не выходят за порог по количеству уникальных ключей, в следствие чего, стадия объединения происходит практически мгновенно даже в худшем случае. На практике данному методу придало ускорение увеличение количества дочерних хеш-таблиц Two-level хеш-таблицы с 256 до 512, так как данные менее плотно распределяются по группам, а значит потоки меньше мешают друг другу блокировками. Еще одна оптимизация, которая на практике дала хороший прирост производительности: осуществлять попытку добавления элемента в дочернюю хеш-таблицу общей только, если элементов из той же группы в буффере накопилось больше 8, что так же снижает конкурентность между потоками. Никаких предварительных вычислений не происходит, агрегация хорошо распараллеливается на любом количестве данных, объедение выполняется мгновенно, поэтому очевидных минусов у данного алгоритма нет.

# 3 Сравнение реализованных подходов

## 3.1 Описание способа тестирования

Для тестирования алгоритмов были использованы несколько наборов реальных анонимизированных данных с разной долей уникальных ключей, предоставленных в онлайн документации ClickHouse [4]. Так же были сгенерированы несколько искусственных наборов, но приоритет отдавался реальным. Для автоматизации тестирования был написан бенчмарк, который запускает различные алгоритмы на разных типах и размерах данных и выводит результаты в читаемом виде. Каждый тест запускался несколько раз с 16, 24 и 32 потоками, из которых бралось лучшее значение времени. Тестирование производилось на сервере с 32 ядрами и 24GB оперативной памяти. Код был написан на языке программирования C++, используемый компилятор – gcc-9 версии 9.3.0, операционная система - Ubuntu 18.04.2 LTS.

Для описания результатов самыми репрезентативными наборами данных являются:

1) WatchID – доля уникальных ключей 99.99%.
2) RefererHash – доля уникальных ключей 21.5%
3) SearchPhrase – доля уникальных ключей 6%

## 3.2 Результаты сравнения для набора данных WatchID

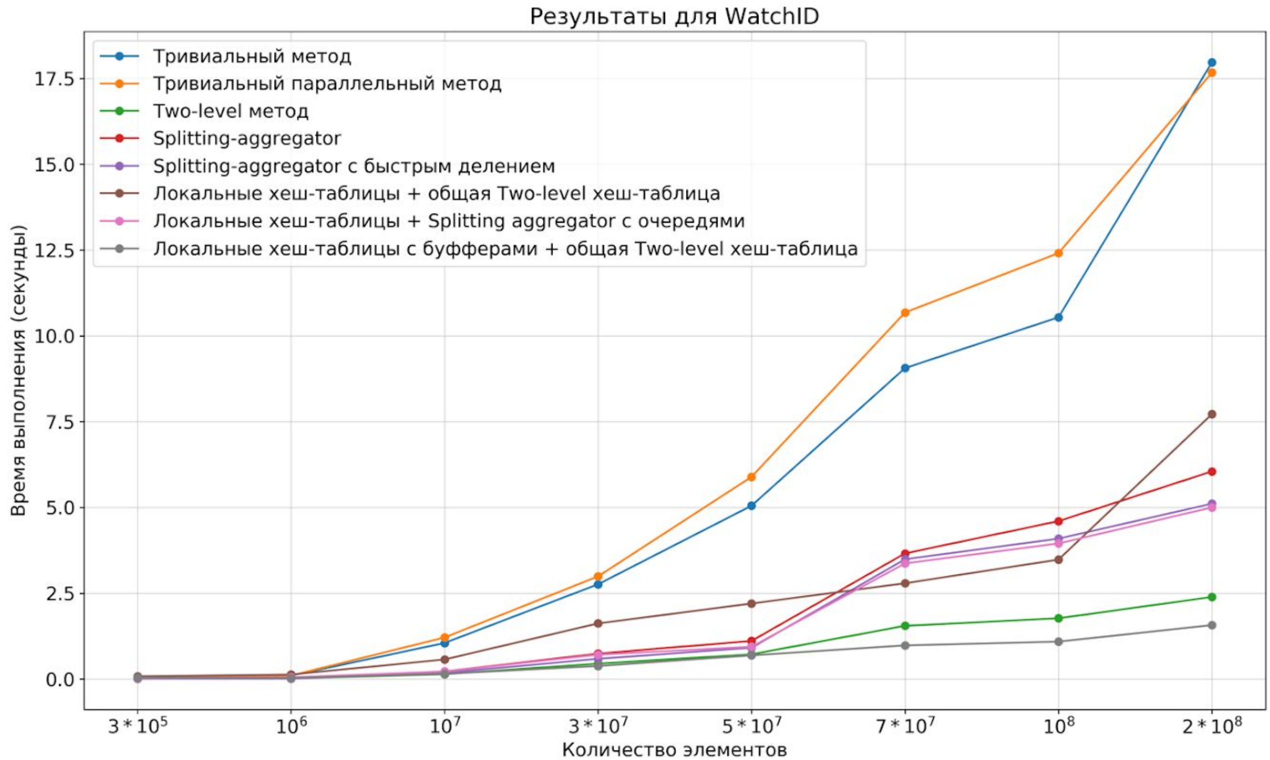


Рис. 3.1. График зависимости времени выполнения алгоритмов от размеров исходных данных для набора данных WatchID

- “Тривиальный метод” и “Тривиальный параллельный метод”: работают очень долго, в несколько раз медленнее остальных методов, причем параллельный метод в основном работал хуже обычного, так как почти все ключи различные, и на стадию объединения уходит много времени.
- “Splitting-aggregator” и “Splitting-aggregator с быстрым делением”: работают довольно быстро, особенно на средних объемах данных до $5 \times 10^7$, в виду большого количества различных ключей. Причем быстрое деление дает ощутимый выигрыш в производительности на большом количестве данных.
- “Локальные хеш-таблицы + общая Two-level хеш-таблица”: работает медленнее “Splitting-aggregator” алгоритмов, так как на очередной итерации локальные таблицы почти не используется, и для каждого ключа делается попытка вставить в общую хеш-таблицу. В итоге

размер локальных хеш-таблиц становится в несколько раз больше порога, что приводит к длительной стадии объединения.

- “Локальные хеш-таблицы + Splitting-aggregator с очередями”: работает примерно столько же, сколько и “Splitting-aggregator с быстрым делением”, в виду большого количества различных ключей.
- “Локальные хеш-таблицы с буфферами + общая Two-level хеш-таблица” и “Two-level метод”: показывают лучшее время, намного обходя остальные методы.

Ниже более подробно рассмотрены последние два метода.

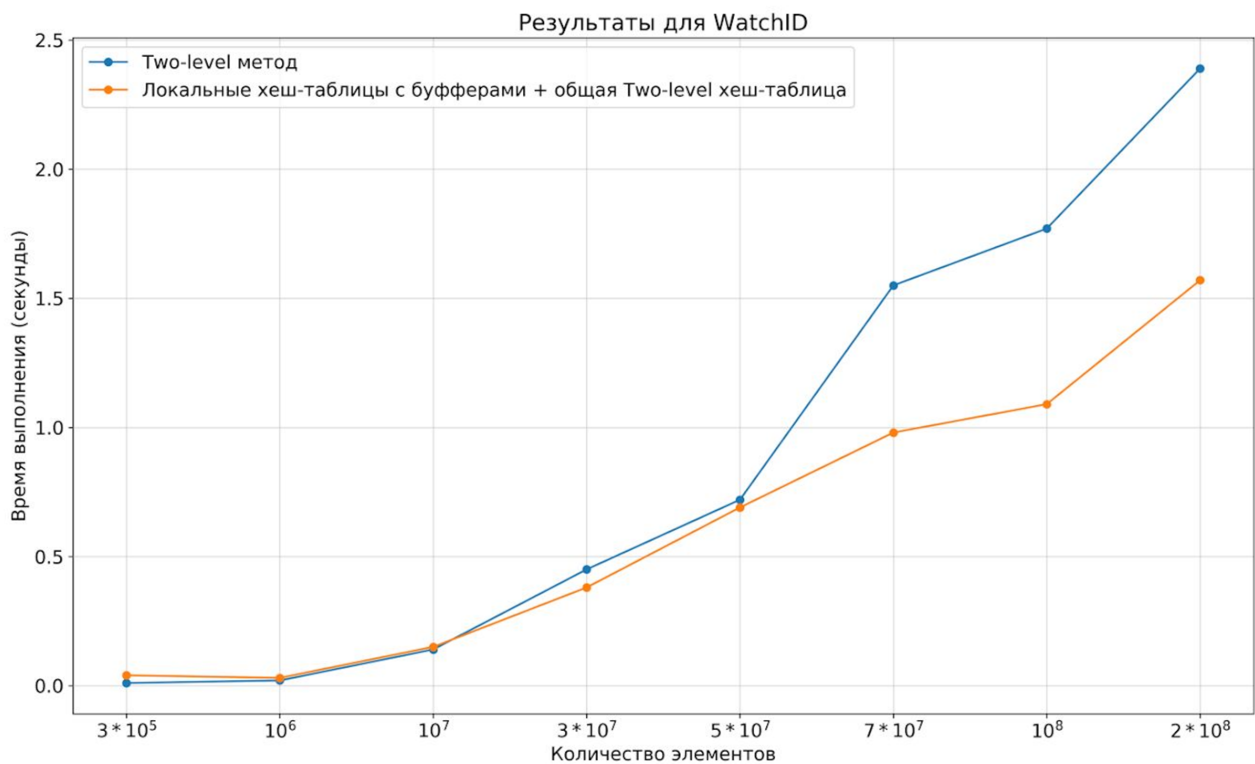


Рис. 3.2. График зависимости времени выполнения алгоритмов “Two-level” и “ Локальные хеш-таблицы с буфферами + общая Two-level хеш-таблица” от размеров исходных данных для набора данных WatchID

Как видно из графика, метод “Локальные хеш-таблицы с буфферами + общая Two-level хеш-таблица” работает значительно быстрее при большом объеме данных.

## 3.3 Результаты сравнения для набора данных RefererHash

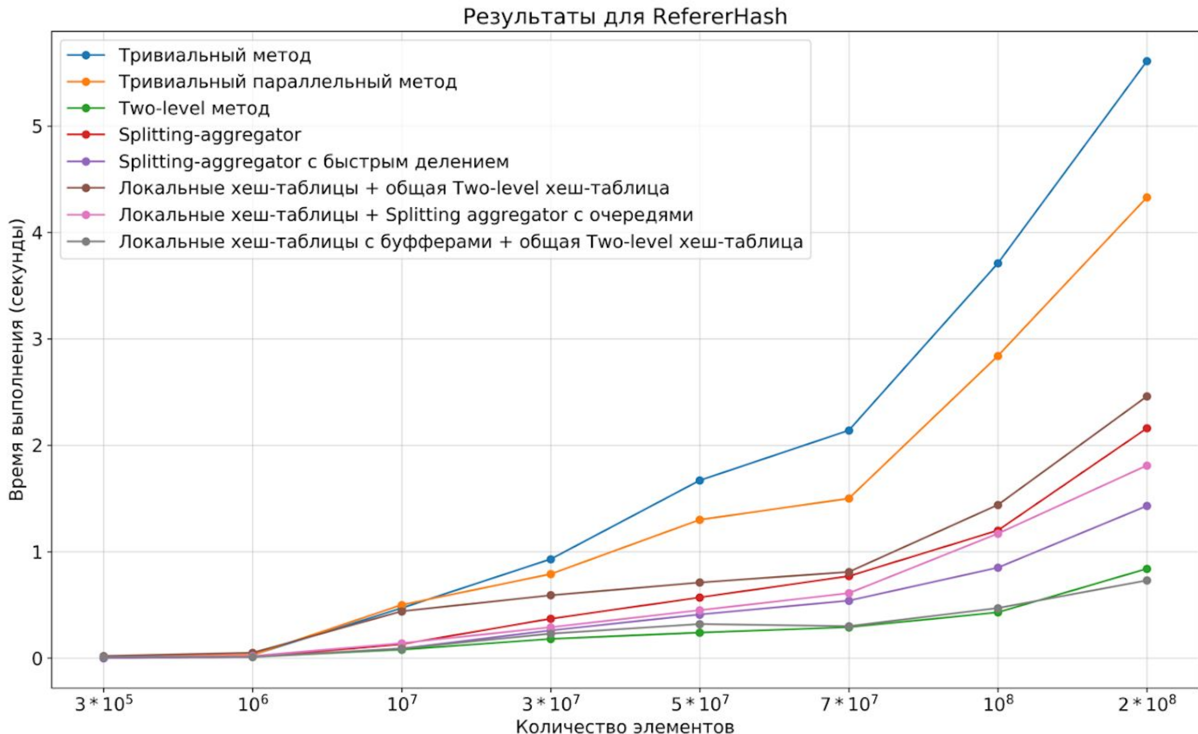


Рис. 3.3. График зависимости времени выполнения алгоритмов от размеров исходных данных для набора данных RefererHash

- "Тривиальный метод" и "Тривиальный параллельный метод": работают гораздо медленнее остальных методов, но не настолько хуже, как для предыдущего набора данных. Параллельный метод в данном случае работает быстрее обычного, так как различных ключей стало в 5 раз меньше.
- "Splitting-aggregator" и "Splitting-aggregator с быстрым делением": в данном случае быстрое деление дает ощутимый выигрыш.
- "Локальные хеш-таблицы + общая Two-level хеш-таблица" и "Локальные хеш-таблицы + Splitting-aggregator с очередями": второй работает ощутимо быстрее первого, но медленнее "Splitting-aggregator с быстрым делением".
- "Локальные хеш-таблицы с буфферами + общая Two-level хеш-таблица" и "Two-level метод": снова показывают лучшее время, но в данном случае разницы в результатах между ними почти нет.

## 3.4 Результаты сравнения для набора данных SearchPhrase

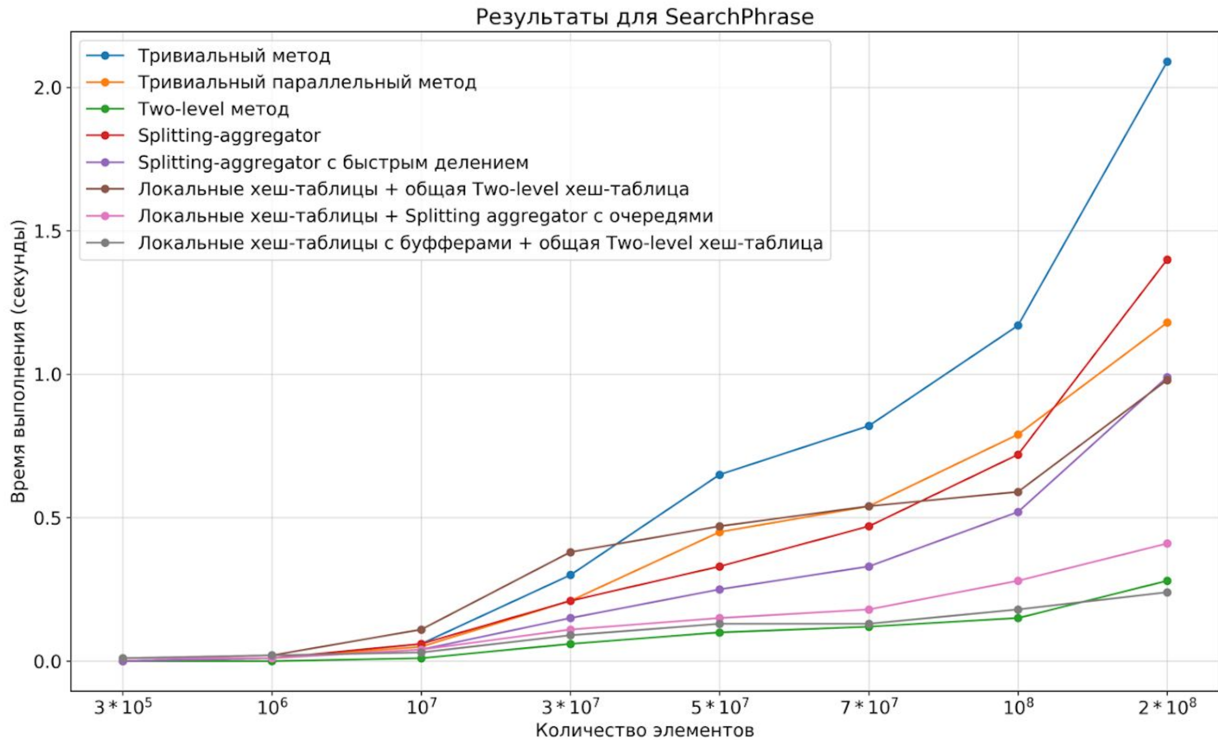


Рис. 3.4. График зависимости времени выполнения алгоритмов от размеров исходных данных для набора данных SearchPhrase

- “Тривиальный метод” и “Тривиальный параллельный метод”: в данном случае работают довольно быстро в абсолютном значении, так как различных ключей всего 5% - происходит мало выделений памяти и эффективно работает оптимизация для подряд идущих одинаковых ключей. Параллельный метод работает быстрее обычного.
- “Splitting-aggregator” и “Splitting-aggregator с быстрым делением”: в данном случае работают примерно столько же, как “тривиальный параллельный метод”, так как большинство потоков завершаются мгновенно, ожидая, пока остальные производят вычисления на своих больших кусках данных. Быстрое деление снова дает выигрыш.
- “Локальные хеш-таблицы + общая Two-level хеш-таблица”: в данном случае общая хеш-таблица используется редко, а значит основное время занимает стадия объединения, которая выполняется в одном потоке, почти так же, как в “Тривиальном параллельном методе”.

- “Локальные хеш-таблицы + Splitting-aggregator с очередями”: показывает очень хорошие результаты, так как это улучшенная версия “Splitting-aggregator” алгоритма, основной идеей которого является решение проблемы малой доли уникальных ключей.
- “Локальные хеш-таблицы с буфферами + общая Two-level хеш-таблица” и “Two-level метод”: снова показывают лучшие результаты и работают примерно одинаковое количество времени.

## 3.5 Выводы

Были реализованы несколько способов параллельной агрегации данных с последующим их сравнением. Каждый из них имеет свои особенности и свою область применения, каждый имеет свои плюсы и минусы. Но самыми эффективными методами в совокупности на различных типах данных оказались:

1) Для маленького количества данных – тривиальный способ с одной хеш-таблицей, так как все методы отрабатывают почти мгновенно, но этот является самым простым в реализации и использует меньше всего памяти.
2) Для большого количества данных, если доля уникальных ключей небольшая (меньше 35%) – “Two-level метод”, алгоритм, который используется в ClickHouse сейчас.
3) Для большого количества данных, если доля уникальных ключей велика (больше 35%) – реализованный новый алгоритм “Локальные хеш-таблицы с буфферами + общая Two-level хеш-таблица”.

Поэтому было принято решение реализовать и применить новый метод “Локальные хеш-таблицы с буфферами + общая Two-level хеш-таблица” в ClickHouse в запросах с GROUP BY для ускорения агрегации на данных с большой долей уникальных ключей.

# 4 Реализация и применение в ClickHouse

## 4.1 Устройство агрегации в ClickHouse

Агрегацию внутри ClickHouse кратко можно описать так. Когда пришел запрос на сервер, он разбирается на отдельные ключевые слова. Если в запросе присутствует выражение GROUP BY, то из него берутся ключи и создаются так называемые процессоры агрегации (объекты класса AggregatingTransform) в количестве равном максимальному количеству потоков, которое ClickHouse может использовать для выполнения запроса – по умолчанию, количество ядер процессора. Каждый процессор агрегации работает в своем потоке. Процессор агрегации содержит в себе свою собственную хеш-таблицу и агрегатор (объект класса Aggregator).

- Блок – часть данных из таблицы. Максимальный размер блока ограничен и по умолчанию равен 65536. Чтение по блокам нужно для того, чтобы сократить потребление оперативной памяти, так как в таблице могут храниться гигабайты данных.
- Агрегатор – это объект, который агрегирует данные из переданного ему блока в переданную ему хеш-таблицу.

Каждый процессор агрегации получает свою часть данных из таблицы по блокам. При получении очередного блока, запускается агрегатор с этим блоком и с текущей хеш-таблицей. Агрегатор, в свою очередь, может конвертировать хеш-таблицу из обычной (далее Single-level) в Two-level, если количество ключей в ней слишком велико, т.е. больше определенного порога. Если пришел пустой блок, то это значит, что данные закончились, и стадия агрегации завершается. В хеш-таблице находятся агрегированные данные из всех полученных за время работы блоков. Далее все хеш-таблицы всех процессоров агрегации объединяются в одну. Это происходит двумя способами: если все хеш-таблицы являются Single-level, то объединение происходит в один поток,

если есть хотя бы одна Two-level, то все оставшиеся конвертируется в Two-level и стадия объединения происходит параллельно в нескольких потоках.

## 4.2 Особенности применения нового метода

Основная сложность применения метода “Локальные хеш-таблицы с буфферами + общая Two-level хеш-таблица” (далее сокращенно новый общий метод) в ClickHouse заключается в том, что в начале выполнения запроса неизвестна доля уникальных ключей, поэтому невозможно полностью убедиться в надобности использования это метода. Поэтому был выбран эвристический способ: после выполнения агрегации над первым блоком, проверяется отношение количества ключей в хеш-таблице к количеству ключей в блоке, и если оно больше определенного порога (например, 0.5, то есть 50% уникальных ключей в блоке), то далее используется новый общий метод. То есть оценка доли уникальных ключей происходит по первому блоку. У данного способа есть очевидный минус: если в начале набора данных находятся только разные ключи, а в оставшейся части повторяющиеся, то новый метод будет использован ошибочно. Простым примером является следующий набор данных: ключи состоят из нескольких групп, в каждой из которых элементы имеют значения от 0 до $k$, где $k$ – число, большее максимального размера блока. То есть, если максимальный размер блока 65536, всего элементов $10^8$, а $k$ равно $10^5$, то для набора данных $0..10^5, 0..10^5, \dots, 0..10^5$, будет использован новый общий метод, хотя доля уникальных ключей в нем составляет всего 0.001%.

Для обеспечения доступа к общей хеш-таблице (используемой в новом общем методе) из разных процессоров агрегации, она создается в единственном экземпляре и передается каждому из них перед началом выполнением агрегации. Вместе с ней создается и передается общий массив из атомарных переменных, которые используются для блокировок дочерних Single-level хеш-таблиц общей хеш-таблицы. То есть процессор агрегации запускает

агрегатор с текущей локальной хеш-таблицей, общей хеш-таблицей и массивом блокировок.

Так как новый общий метод используется только для данных, с большой долей уникальных различных ключей, то для оптимизации именно под этот случай на каждой итерации алгоритма не делается поиск очередного ключа в локальной хеш-таблице, а сразу осуществляется попытка вставки в общую хеш-таблицу. Оставшиеся в буффере ключи по-прежнему вставляются в локальную хеш-таблицу после всех итераций. Так же стоит отметить, что из-за того, что агрегация данных происходит по блокам, а не разом на всем множестве, на протяжении работы всей агрегации буффер очищается и ключи сбрасывается в локальную хеш-таблицу несколько раз, а не один, как в базовой версии нового общего метода, но это не приводит к критичному увеличению ее размера.

Еще одна проблема заключается в следующем. Процессоры агрегации работают независимо друг от друга, поэтому для одного из них может наступить момент, когда начинает использоваться новый общий метод, а для других нет, поэтому на выходе от разных процессоров агрегации могут быть разные типы хеш-таблиц, которые в последствии следует конвертировать к одному виду. На это уходит какое-то время, но не критичное в рамках запроса.

С точки зрения производительности, лучшим количеством дочерних хеш-таблиц для общей хеш-таблицы оказалось значение 2048. Порог доли уникальных ключей и минимального количества элементов в буффере для определенной дочерней хеш-таблицы были вынесены в параметры, которые пользователь может при желании поменять. Их значения по умолчанию 0.5 и 20 соответственно.

Финальный способ применения можно описать так:

1. Стадия агрегации. Каждый процессор агрегации независимо в своем потоке:
   a. Выполняет агрегацию первого блока в Single-level таблицу
   b. Если доля уникальных ключей больше определенного порога (по умолчанию 0.5), то конвертирует хеш-таблицу в тип общей хеш-таблицы и начинает использовать новый общий метод. Агрегация происходит с использованием локальной и общей хеш-таблицы, пока блоки не закончатся.
2. Стадия объединения: потоки параллельно объединяют дочерние хеш-таблицы с одинаковыми номерами полученных Two-level хеш-таблиц, включая общую, в одну (по умолчанию дочерних хеш-таблиц у общей 2048). Данная стадия происходит почти мгновенно, если во всех процессорах агрегации используется новый общий метод.

## 4.3 Тестирование

Код был написан на языке программирования C++ с помощью интегрированной среды разработки CLion. Для отладки использовался отладчик lldb.

Для проверки работоспособности новой функциональности было проведено ручное и автоматическое тестирование во внутренней системе непрерывной интеграции Яндекса. В автоматические тесты входят следующие типы: функциональные, на производительность, компилируемость, форматирование кода, интеграционные и множество других. Все тесты кроме производительных прошли без изменений, то есть были пройдены. Тесты на производительность ожидаемо показали ускорение на запросах с большой долей уникальных ключей, и замедление на запросах с малой долей уникальных ключей, в которых данные имеют не обычное распределение, в следствие чего новый метод используется неправильно. Основные из них:

Таблица 4.1. Результаты тестов на производительность

| Запрос | Время выполнения без нового метода, в секундах | Время выполнения с новым методом, в секундах | Абсолютное ускорение, в секундах | Относительное ускорение |
|---|---|---|---|---|
| SELECT number % toUInt32(1e8) AS k, count() FROM numbers_mt(toUInt32(1e8)) GROUP BY k FORMAT Null | 1.8195 | 1.0905 | -0.729 | -0.401 |
| SELECT number % 10000000 AS k, count() FROM numbers_mt(80000000) GROUP BY k FORMAT Null | 0.9153 | 0.6163 | -0.299 | -0.327 |
| SELECT number % 1000000 AS k, count() FROM numbers_mt(160000000) GROUP BY k FORMAT Null | 0.5268 | 0.6592 | 0.1324 | 0.251 |
| SELECT number % 100000 AS k, count() FROM numbers_mt(160000000) GROUP BY k FORMAT Null | 0.1932 | 0.3991 | 0.2059 | 1.065 |

В запросах с 100% и 12.5% долями уникальных ключей было достигнуто существенное ускорение с помощью нового общего метода. В запросах с 0.625% и 0.0625% долями уникальных ключей наблюдаются замедления, которые в абсолютных значениях менее значительны, но в относительных значениях довольно велики.

## 4.4 Выводы

Новый общий метод был успешно реализован и применен в ClickHouse. Он продемонстрировал существенное ускорение операции GROUP BY в ClickHouse на данных, с большой долей уникальных ключей. Но из-за проблемы с определением целесообразности его использования, он немного замедляет выполнение более простых запросов с малой долей уникальных ключей, если их распределение не является обычным. Данную проблему

возможно решить, если в процессоре агрегации будет информация о примерном общем количестве элементов, что позволит определять долю уникальных ключей не по первому блоку, а на определенном фиксированном проценте от всех данных. Данную функциональность предстоит реализовать в дальнейшем.

Ссылка на созданный pull-request в репозиторий ClickHouse на github.com: https://github.com/ClickHouse/ClickHouse/pull/10956/

# 5 Заключение

В наше время работа с большими данными является неотъемлемой частью жизни IT-компаний, работающих в них аналитиков и разработчиков. Различных данных, метрик и логов становится все больше, и для работы с ними нужен подходящий инструмент, который предоставит возможность их хранить в большом объеме и эффективно обрабатывать. Одним из самых популярных таких инструментов является СУБД ClickHouse, в виду своей высокой эффективности работы с данными.

Одной из базовых возможностей таких систем является агрегация данных по ключу, которая в ClickHouse выполняется с помощью эффективного параллельного алгоритма. В данной работе были предложены и реализованы альтернативные новые параллельные способы агрегации. Для каждого из них было произведено тестирование с различными параметрами: тип набора данных, их количество, количество потоков. На полученных результатах тестирования был произведен сравнительный анализ методов, были определены преимущества и недостатки каждого из них.

Лучшие результаты показал новый разработанный алгоритм “Локальные хеш-таблицы с буфферами + общая Two-level хеш-таблица”. Он сочетает в себе идею использования Two-level хеш-таблицы и массового выполнения операций из *flat combining*. Данный алгоритм для наборов данных с малой и средней долями уникальных ключей работает примерно с такой же эффективностью, как метод, используемый в ClickHouse сейчас. Но если доля уникальных ключей велика, то он показывает существенное увеличение производительности.

Данный метод был адаптирован и реализован в ClickHouse в виде pull-request в репозиторий на github.com. Было произведено ручное и автоматическое тестирование, которые подтверждают правильность и корректность реализации. Тесты на производительность показали существенное

ускорение на наборах данных с большой долей уникальных ключей. Тем не менее, была обнаружена проблема с правильность применения данного алгоритма, которая обуславливается отсутствием информации о количестве уникальных ключей перед выполнением запроса, приводящая к небольшому в абсолютном значении, но ощутимому в относительном, замедлению на определенных типах запросах с малой долей уникальных ключей. Для решения данной проблемы был предложен способ, который в дальнейшем возможно реализовать.

Идеи из предложенных в работе алгоритмов могут быть в дальнейшем доработаны и усовершенствованы для создания новых алгоритмов параллельной агрегации.

# 6 Источники


1. Clickhouse.tech. [Электронный ресурс] // *Синтаксис запросов SELECT*. URL: https://clickhouse.tech/docs/en/query_language/select/#select-group-by-clause/ (дата обращения: 01.02.2020).
2. Daniel Lemire's blog. [Электронный ресурс] // *A Fast Alternative To The Modulo Reduction*. URL: https://lemire.me/blog/2016/06/27/a-fast-alternative-to-the-modulo-reduction/ (дата обращения: 20.04.2020).
3. D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. ACM, 2010.
4. Clickhouse.tech. [Электронный ресурс] // Анонимизированные данные Яндекс.Метрики. URL: https://clickhouse.tech/docs/en/getting-started/example-datasets/metrica/ (дата обращения: 21.04.2020).