

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

Факультет компьютерных наук
Основная образовательная программа
Прикладная математика и информатика

КУРСОВАЯ РАБОТА

ПРОГРАММНЫЙ ПРОЕКТ

НА ТЕМУ

**"МОДИФИКАТОРЫ DISTINCT И ORDER BY ДЛЯ ВСЕХ
АГРЕГАТНЫХ ФУНКЦИЙ"**

Выполнил студент группы 175, 3 курса,
Борзенкова София Сергеевна

Руководитель КР:
Руководитель группы разработки,
Миловидов Алексей Николаевич

Москва 2020

Содержание

1	Введение	5
1.1	СУБД ClickHouse	5
1.2	Актуальность и значимость	5
1.3	Постановка задачи	5
1.4	Цель и задачи	6
1.5	Основной результат	6
1.6	Структура работы	6
2	Существующие решения	7
2.1	Комбинаторы в ClickHouse	7
2.2	Модификаторы в других СУБД	8
2.2.1	PostgreSQL	8
2.2.2	MySQL	8
2.2.3	Apache Druid	9
2.2.4	Другие БД	10
2.3	Выводы	10
3	Архитектура ClickHouse	11
3.1	Обработка запросов	11
3.2	Агрегатные функции	12
3.2.1	Архитектура	12
3.2.2	Исполнение	13
3.2.3	Данные	13
3.3	Комбинаторы агрегатных функций	13
3.3.1	Что такое комбинаторы	13
3.3.2	Отличия от агрегатных функций	14
3.3.3	Обработка данных	14
3.3.4	Методы	14
3.4	Модификатор <i>DISTINCT</i>	14

3.5	Модификатор <i>ORDER BY</i>	15
3.6	Выводы и результаты	15
4	Решение	16
4.1	Структура	16
4.1.1	Архитектура	16
4.1.2	Хранение данных	16
4.1.3	Изменение состояния	17
4.1.4	Состояние комбинатора	17
4.1.5	Состояние вложенной агрегатной функции	17
4.2	Модификатор DISTINCT	18
4.2.1	Аргументы	18
4.2.2	Элементы класса	18
4.2.3	Инициализация	18
4.2.4	Данные	19
4.2.5	Метод <i>add</i>	19
4.2.6	Метод <i>merge</i>	20
4.3	Модификатор ORDER BY	20
4.3.1	Изменения в архитектуре	20
4.3.2	Аргументы	21
4.3.3	Параметры	21
4.3.4	Элементы класса	22
4.3.5	Инициализация	22
4.3.6	Данные	23
4.3.7	Метод <i>add</i>	23
4.3.8	Метод <i>insertResultInto</i>	24
4.4	Выводы и результаты	24
5	Проверка результатов	25
5.1	Тестирование	25
5.2	Функциональные тесты	25

5.2.1	Функциональные тесты в ClickHouse	25
6	Заключение	26
6.1	Результаты	26
6.2	Развитие	26
7	Источники	26
8	Приложения	28
8.1	Глоссарий	28
8.2	Список сокращений	29
8.3	Эксперименты	29
8.3.1	Ручное тестирование	29
8.3.2	Функциональные тесты	30

Аннотация

В этой работе будет рассмотрена реализация новых компонент СУБД ClickHouse компании «Яндекс». Пользователи могут создавать любые валидные SQL запросы, и база данных должна уметь их обрабатывать. На данный момент в ClickHouse реализована не вся возможная функциональность, однако разработчики всячески стараются это исправить. В этой работе будут рассмотрены два дополнения к существующей функциональности – агрегатным функциям, позволяющим пользователям получать собирательные статистики вместо данных в исходном виде. В результате работы реализованы модификаторы *DISTINCT* и *ORDER BY*, позволяющие изменять количество и порядок данных перед попаданием в агрегатную функцию.

Abstract

In this paper we will consider realization of some new components in ClickHouse DBMS. Users can create any valid SQL queries to database and it should be able to handle them. Now ClickHouse doesn't support some features, but developers are trying to fix it. In this work we will consider two extentions to aggregate functions in ClickHouse. Aggregate functions allow users to get data statistics instead of collecting raw data. As a result we implemented modifiers *DISTINCT* and *ORDER BY*, which allow to change amount and order of data before it passes into respective aggregate function.

Ключевые слова

Модификатор, агрегатная функция, реализация, SQL, СУБД.

1 Введение

1.1 СУБД ClickHouse

ClickHouse – система управления базами данных (СУБД), разрабатываемая компанией «Яндекс». Пользовательское взаимодействие с данными в ней происходит при помощи языка SQL. СУБД написана на языке C++ и имеет открытый исходный код на платформе GitHub [1]. Вся разработка в рамках работы велась в указанной кодовой базе.

1.2 Актуальность и значимость

Как известно, у языка SQL большое количество возможностей, однако не все они на данный момент поддерживаются в ClickHouse. В этой работе будет рассматриваться реализация обработки запросов на языке SQL, подразумевающих предобработку данных модификаторами *DISTINCT* и *ORDER BY*. На момент выполнения работы в ClickHouse не была реализована их поддержка, однако некоторые пользователи хотели иметь возможность строить такие запросы в рамках возможностей языка SQL. Так как ряд других СУБД поддерживает запросы, использующие *DISTINCT* или *ORDER BY*, ClickHouse мог терять потенциальных пользователей. Реализация схожей функциональности в других СУБД также будет рассмотрена в работе.

1.3 Постановка задачи

Неформальная постановка задачи: необходимо реализовать поддержку запросов, содержащих агрегатные функции с модификатором *DISTINCT* или *ORDER BY*. С формальной точки зрения, в задаче требуется реализовать два комбинатора агрегатных функций в текущей архитектуре ClickHouse.

1.4 Цель и задачи

Целью работы является расширение функциональности СУБД ClickHouse. В рамках работы стояли следующие задачи:

1. Изучить архитектуру ClickHouse, имеющую отношение к агрегатным функциям.
2. Реализовать в рамках отдельных комбинаторов агрегатных функций модификаторы *DISTINCT* и *ORDER BY* и подключить их к фабрике комбинаторов.
3. Провести тестирование реализованной функциональности.
4. Добавить функциональные тесты для написанного кода.

1.5 Основной результат

Результатом работы является внедрение в текущую архитектуру модификаторов *DISTINCT* и *ORDER BY* в рамках комбинаторов агрегатных функций ClickHouse, а также написание автоматических тестов для новой функциональности. Таким образом, любая агрегатная функция может иметь суффикс *—Distinct* или *—OrderBy* с соответствующим изменением её аргументов, параметров и входных данных.

1.6 Структура работы

В главе 2 рассмотрены СУБД, в которых реализована схожая с поставленной задачей или аналогичная ей функциональность. В ней более детально рассмотрены комбинаторы в ClickHouse (разд. 2.1), агрегатные функции в PostgreSQL (разд. 2.2.1) и MySQL (разд. 2.2.2), а также комбинаторы в Apache Druid (разд. 2.2.3). Далее, в главе 3 подробнее рассмотрена относящаяся к задаче архитектура СУБД ClickHouse: описан процесс обработки запросов, содержащих агрегатные функции, устройство агрегатных функций

и их комбинаторов. Затем, в главе 4 описана основная идея решения задачи: общая структура (разд. 4.1), а также подробности реализации комбинаторов *–Distinct* (разд. 4.2) и *–OrderBy* (разд. 4.3) в рамках имеющейся архитектуры. В главе 5 рассмотрены проведенные тестирования новой функциональности. Далее, в главе 6 описаны полученные результаты и предложены дальнейшие перспективы развития функциональности, реализованной в работе. Глава 7 содержит список интернет-источников, использованных при написании работы. Наконец, в главе 8 можно найти приложения: глоссарий, список сокращений и проведённые эксперименты.

2 Существующие решения

2.1 Комбинаторы в ClickHouse

В ClickHouse поддерживаются некоторые комбинаторы для агрегатных функций [2] (например, *If*, *Array*, *Merge* и другие). Они модифицируют работу агрегатных функций, к которым применяются. Например, комбинатор *If* изменяет количество аргументов функции: она принимает дополнительный аргумент – условие, по которому нужно отбирать подходящие строки. Некоторые комбинаторы изменяют тип принимаемых агрегатной функцией аргументов или её возвращаемого значения (например, модификатор *Array* меняет аргументы функции с T на $Array(T)$, а *State* меняет возвращаемого значения на $AggregateFunction(...)$). Эта структура близка к рассматриваемой задаче, так как реализации комбинаторов задействуют одинаковую архитектуру. Однако на момент выполнения работы среди имеющихся модификаторов не существовало подходящих по логике к рассматриваемым в задаче, поэтому считать задачу решенной, имея только существующие комбинаторы, было нельзя. Тем не менее, общая архитектура комбинаторов во многом используется в решении задачи. Также при решении задачи используются уже имеющиеся в ClickHouse структуры для уникализации и сортировки данных.

2.2 Модификаторы в других СУБД

2.2.1 PostgreSQL

В СУБД PostgreSQL также есть агрегатные функции и их модификаторы [3]. Однако, в отличие от ClickHouse, в ее реализации нет комбинаторов, которые можно сочетать с агрегатными функциями. Вызов агрегатной функции происходит с помощью структуры *CallStmt*, два аргумента которой – *FuncExpr* и *FuncCall* – отвечают за функцию и ее параметры. В структуре *FuncCall* [4] содержатся, помимо прочих, аргументы *List * agg_order* и *bool agg_distinct*. Параметр *agg_distinct = True* обозначает, что аргументы агрегатной функции помечаются *DISTINCT* и далее обрабатываются с учетом этого условия (уникализируются в рамках агрегатной функции). Вторым параметром, *agg_order ≠ null*, обозначает, что к аргументам функции добавляется условие *ORDER BY* с перечислением элементов списка параметра *agg_order*. Эти аргументы далее также обрабатываются внутри агрегатной функции, сортируя данные по заданным столбцам. Таким образом, PostgreSQL имеет схожую с задачей функциональность, но отличную от ClickHouse архитектуру агрегатных функций. В ClickHouse модификаторы комбинируются с агрегатными функциями, некоторым образом изменяя их выполнение, в то время как в PostgreSQL для каждого модификатора существует свой параметр, проверяющийся на *true* или *not null* при вызове функции. Такая архитектура не подходит для решения поставленной задачи, однако реализация преобразований *DISTINCT* и *ORDER BY* (уникализация для *DISTINCT* и сортировка для *ORDER BY*) в силу ее простоты, частично совпадает с используемой в решении задачи.

2.2.2 MySQL

В СУБД MySQL выполнение агрегатных функций архитектурно устроено схожим с PostgreSQL образом. Однако агрегатные функции вызываются не одинаковым образом из одной структуры как в PostgreSQL: каждая

функция вызывается из структуры своей группы. Таким образом, существует класс *Item_result_field* [5], который задействует класс группы. Функции делятся на группы в соответствии со своей функциональностью (*Item_func*, *Item_sum* и другие). В классе группы содержатся поля, необходимые для исполнения агрегатной функции. Эти поля заполняются классом, который вызывает данный. В классах групп, логика которых предусматривает использование модификаторов *DISTINCT* и *ORDER BY*, есть поля, обозначающие наличие/отсутствие конкретного модификатора (например, поле-функция *bool has_with_distinct()* в классе *Item_sum* [6] значит наличие модификатора *DISTINCT* для вызова функции в случае, если ее значение равно *true*, или его отсутствие в случае, если оно *false*). В свою очередь класс группы вызывает конкретную функцию с заданными параметрами. Если *has_with_distinct = true*, функция выполняется, используя агрегатор *Aggregator_distinct* [7] – класс, который наследуется от интерфейса *Aggregator* [8] и уникализирует входящие данные. Аналогично PostgreSQL архитектура MySQL не схожа с архитектурой ClickHouse, несмотря на наличие отдельных агрегатных функций для *distinct*, в связи с чем такое решение также не подходит для рассматриваемой задачи. Реализация преобразований, выполняемых *Aggregator_distinct*, как и упомянутых преобразований в PostgreSQL, схожа с такой же функциональностью в решении задачи.

2.2.3 Apache Druid

Apache Druid – еще одна рассмотренная база данных. В отличие от всех уже перечисленных, она написана на языке Java. Отчасти она архитектурно схожа с ClickHouse: в её реализации также присутствуют агрегатные функции [9] и комбинаторы, которые можно совмещать с агрегатными функциями. Однако искомым модификаторов в её реализации на данный момент также нет. Например, *count(DISTINCT...)* является составляющей реализации функции *count* [11]. Подсчет уникальных значений в этой базе данных выполняет агрегатная функция класса *HyperUniquesAggregator* [12] (ана-

логично функции *uniqExact* в ClickHouse). То есть отдельного комбинатора для *DISTINCT* в Druid тоже нет. Также аналогично ClickHouse в Druid есть реализация *ORDER BY* только для организации выходных данных, но не для упорядочивания аргументов агрегатных функций [13]. Таким образом, несмотря на то, что архитектурно база данных Druid схожа с ClickHouse, в её реализации нет новых, подходящих для выполнения задачи, идей.

2.2.4 Другие БД

Схожая функциональность также присутствует в том или ином виде и в других базах данных (например, в MonetDB, в которой реализация *DISTINCT* схожа с PostgreSQL [14]), однако в них либо нет агрегатных функций, либо их выполнение, как и в некоторых рассмотренных БД, отличается от представленного в ClickHouse. Поэтому их архитектурное решение также не позволяет использовать реализацию этой функциональности в ClickHouse.

2.3 Выводы

Рассмотрев существующие решения в ClickHouse и других базах данных, можно убедиться, что использовать их реализацию для решения поставленной задачи нельзя. Во многих СУБД присутствует схожая с поставленной задачей функциональность – агрегатные функции, однако их архитектура не схожа с ClickHouse (отсутствуют комбинаторы, а модификаторы *DISTINCT* и *ORDER BY* используются как параметры вызова агрегатных функций), поэтому не может быть использована. В базе данных Apache Druid, реализация которой схожа с ClickHouse за исключением языка программирования, также есть агрегатные функции и комбинаторы для них, однако функциональность развита на таком же уровне, что и у ClickHouse, то есть новых идей для выполнения задачи также не было обнаружено. В самом ClickHouse уже существует схожая с требуемой функциональность, однако на момент вы-

полнения работы она исполняла задачи, отличные от поставленной. В рамках задачи добавлены модификаторы агрегатных функций *DISTINCT* и *ORDER BY* в имеющуюся архитектуру ClickHouse по аналогии с другими, уже функционирующими, комбинаторами. На примере рассмотренных СУБД также можно сказать, что логика реализации конкретных функций *DISTINCT* и *ORDER BY* является схожей для всех баз данных – модификатор *DISTINCT* при помощи хэш-таблицы обрабатывает данные и возвращает уникальные, а *ORDER BY* сортирует данные по заданным столбцам и возвращает в некоторых базах данных только запрошенные столбцы, в других – все (для дальнейшей обработки с отбором столбцов). Аналогично реализованы модификаторы *DISTINCT* и *ORDER BY* в ClickHouse.

3 Архитектура ClickHouse

3.1 Обработка запросов

В языке SQL есть запросы, содержащие в себе агрегатную функцию. Например, в запросе *SELECT count(value) FROM table_name* ключевое слово *count* – агрегатная функция, возвращающая количество строк в столбце *value* таблицы *table_name*. Запросы, поступающие на сервер ClickHouse, после предобработки распределяются на исполняющие элементы. За исполнение агрегатных функций отвечает фабрика агрегатных функций. В соответствии с указанным в запросе именем функции она создает экземпляр подходящего класса. Далее данные обрабатываются этим классом в несколько потоков исполнения, после чего совмещаются и присоединяются к будущему ответу на запрос.

3.2 Агрегатные функции

3.2.1 Архитектура

Агрегатные функции [15] собирают статистику данных для запроса. Они представляют собой классы вида *AggregateFunction*<Name>, наследующиеся от базового класса-интерфейса *IAggregateFunction* при помощи класса *IAggregateFunctionHelper* или *IAggregateFunctionDataHelper* в соответствии со схемой на рис. 3.1. В агрегатной функции, наследованной от ин-

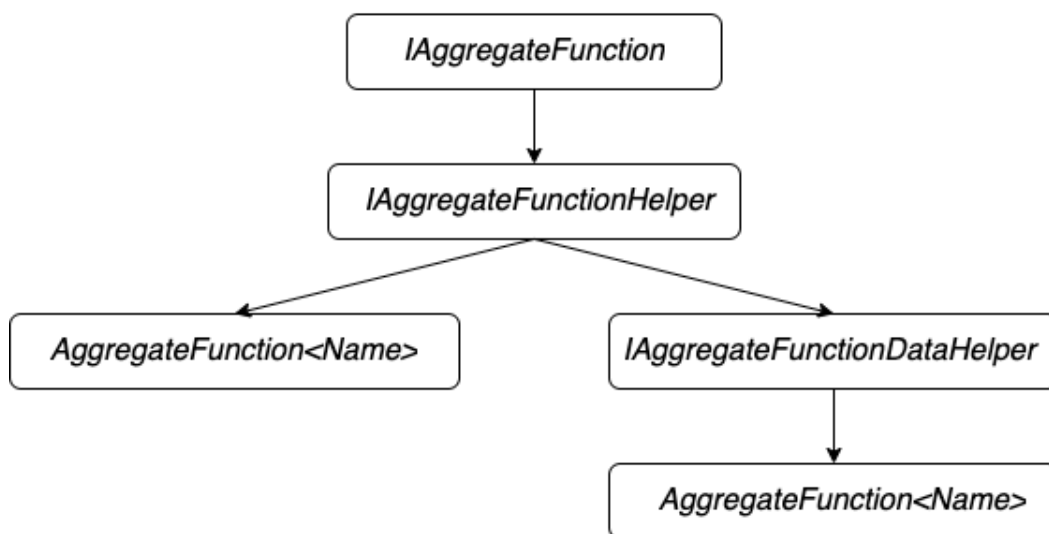


Рис. 3.1. Архитектура наследования агрегатных функций

терфейса *IAggregateFunctionDataHelper*, можно управлять данными из её состояния, используя метод *data* этого интерфейса. Агрегатные функции работают по идеологии MapReduce - данные делятся на блоки, далее каждый поток исполнения применяет метод *add* для каждой строки своего блока, а затем происходит *merge* результатов, полученных каждым процессом, и присоединение финального ответа к колонкам ответа на запрос. Интерфейс *IAggregateFunction* содержит большое количество методов, однако в рамках задачи в основном рассматриваются методы *add*, *merge* и *insertResultInto*, а также некоторые методы для управления памятью.

3.2.2 Исполнение

За исполнение агрегатных функций отвечает фабрика агрегатных функций. Для регистрации в ней в реализации агрегатной функции существует функция *registerAggregateFunction<Name>* вне основного класса. Регистрация происходит с помощью функции *createAggregateFunction<Name>*, в которой выполняется проверка аргументов и инициализация функции.

3.2.3 Данные

Многим агрегатным функциям необходимо хранить какие-либо данные для последующей обработки и выдачи ответа. Такие функции наследуются от класса *IAggregateFunctionDataHelper*. Для хранения данных в агрегатных функциях есть состояние. Оно передается во все методы, в которых может понадобиться, как аргумент *AggregateDataPtr place*. С помощью вызова метода *data(place)* базового класса можно получить экземпляр структуры данных, хранящийся в *place*, и далее взаимодействовать с ним.

3.3 Комбинаторы агрегатных функций

3.3.1 Что такое комбинаторы

Некоторые агрегатные функции можно комбинировать с другими – в архитектуре ClickHouse такие функции называются комбинаторами. Например, для запроса *SELECT count(DISTINCT value) FROM table_name* ключевое слово *DISTINCT* – модификатор, которому соответствует комбинатор – *Distinct*, для агрегатной функции *count*, возвращающий выборку уникальных строк в столбце *value*). Комбинаторы представляют собой агрегатные функции, классы которых, помимо прочих необходимых конкретной функции полей, обязательно содержат поле *AggregateFunctionPtr nested_func* – функцию, с которой комбинируется данная.

3.3.2 Отличия от агрегатных функций

Комбинаторы имеют несколько ключевых отличий от обычных агрегатных функций. Во-первых, их регистрация происходит в другой фабрике – фабрике комбинаторов агрегатных функций. Во-вторых, для создания экземпляра вместо функции *createAggregatorFunction<Name>* они используют класс *AggregateFunctionCombinator<Name>*, наследующийся от класса-интерфейса *IAggregateFunctionCombinator*. Основные методы этого интерфейса изменяют поданные аргументы, параметры или агрегатную функцию. Наконец, методы агрегатной функции-комбинатора не агрегируют данные, а модифицируют их при необходимости и далее вызывают методы вложенной агрегатной функции (*nested_func*).

3.3.3 Обработка данных

При выполнении запроса данные вначале обрабатываются комбинатором, откуда попадают (или, в некоторых случаях, не попадают) во вложенную агрегатную функцию. В некоторых случаях комбинаторы хранят в своем состоянии промежуточное состояние данных для дальнейшей обработки.

3.3.4 Методы

Комбинатор, который ничего не делает, имеет в каждом методе вызов этого же метода вложенной агрегатной функции с такими же параметрами. Функциональность комбинатора добавляется переопределением одного или нескольких методов.

3.4 Модификатор *DISTINCT*

Модификатор *DISTINCT* изменяет данные, передаваемые в агрегатную функцию - принимая все данные, он возвращает только уникальные по заданным столбцам. На момент начала выполнения работы этот модификатор был

применим только к агрегатной функции *COUNT*. Он был реализован как составляющая этой агрегатной функции – в случае наличия в запросе суффикса *–Distinct* вместо функции *countDistinct* создавалась функция *uniqExact*. Эта функция является полноценной агрегатной функцией, поэтому не может быть применима как комбинатор для других агрегатных функций. Более того, она считает количество уникальных значений, а не уникализированные данные, что также делает невозможным любое её использование вместо модификатора *DISTINCT* для всех агрегатных функций.

3.5 Модификатор *ORDER BY*

Модификатор *ORDER BY* упорядочивает входные данные по заданным столбцам и передает указанные для передачи столбцы агрегатной функции. Например, в выражении *groupBy(x ORDER BY y, z)* в агрегатную функцию *groupBy* подаются строки столбца *x* из данных, упорядоченных по столбцам *y* и *z*). На момент начала выполнения работы в ClickHouse этот модификатор не был реализован.

3.6 Выводы и результаты

Из имеющейся архитектуры были сделаны выводы о работе над задачей. В первую очередь, ясно, что модификаторы *DISTINCT* и *ORDER BY* ранее не были реализованы и нуждаются в полном продумывании идеи. Также понятно, что реализовывать модификаторы нужно в рамках комбинаторов агрегатных функций, используя наследование от класса-интерфейса *IAggregateFunctionDataHelper* для доступа к состоянию функции для хранения данных.

4 Решение

4.1 Структура

4.1.1 Архитектура

По итогам главы 3 было определено, что модификаторы *DISTINCT* и *ORDER BY* должны быть реализованы в рамках комбинаторов агрегатных функций. Таким образом, оба модификатора представлены классами *AggregateFunctionDistinct* и *AggregateFunctionOrderBy* имеют структуру интерфейса *IAggregateFunction* с добавлением дополнительных возможностей класса *IAggregateFunctionDataHelper*. В ходе более подробного рассмотрения каждого из комбинаторов, внимание будет акцентировано на трех методах этого интерфейса – *add*, *merge* и *insertResultInto*, так как остальные методы построены аналогично другим комбинаторам. Также были переопределены некоторые методы для управления памятью, их архитектура будет рассмотрена далее в этом разделе.

4.1.2 Хранение данных

Агрегатные функции позволяют хранить необходимые данные в состоянии. Так как комбинаторы являются агрегатными функциями, в их состоянии также можно хранить данные. Однако, нужно учитывать, что состояние комбинатора и его вложенной агрегатной функции хранятся в одном блоке памяти, который выделяется в методе *create*, а освобождается в методе *destroy*. Таким образом, в *place* комбинатора должны храниться два состояния: комбинатора и агрегатной функции. Для осуществления этого было добавлено поле *prefix_size*, функции *getNestedPlace* приватного доступа для преобразования *place* типов *AggregateDataPtr* и *ConstAggregateDataPtr*, а также переопределены методы *create*, *destroy* и *sizeOfData* комбинатора.

4.1.3 Изменение состояния

Поле *prefix_size* отвечает за размер сдвига блока памяти, необходимый для хранения состояния комбинатора, и инициализируется в конструкторе комбинатора размером структуры данных, используемой в комбинаторе. Функции *getNestedPlace* принимают аргумент *place* типа *AggregateDataPtr* или *ConstAggregateDataPtr* и возвращают *place + prefix_size* того же типа, то есть освобождают первые *prefix_size* блоков памяти *place* под хранение состояния комбинатора. Метод *sizeOfData* возвращает сумму размеров состояний комбинатора и вложенной агрегатной функции: *prefix_size + nested_func->sizeOfData()*.

4.1.4 Состояние комбинатора

Состояние комбинатора создается в методе *create* комбинатора исполнением *new (place) Data* и хранится в первых *prefix_size* блоках *place*. Доступ к его данным можно получить, вызвав метод *data(place)* базового класса. В этом случае структура, являющаяся классом *Data* комбинатора, создается из *place*, используя первые *prefix_size* необходимых ей для инициализации блоков памяти. Состояние удаляется в методе *destroy* комбинатора с помощью вызова деструктора структуры хранения данных: *data(place).~Data()*. Под *Data* подразумевается структура данных, используемая в конкретном комбинаторе.

4.1.5 Состояние вложенной агрегатной функции

Состояние вложенной агрегатной функции создается в методе *create* комбинатора с помощью вызова одноименного метода у класса этой функции. Однако вместо *place*, полученного методом комбинатора, в этом вызове используется *getNestedPlace(place)*. Аналогично при всех вызовах методов вложенной агрегатной функции, нуждающихся в передаче *place*, происходит вызов метода *getNestedPlace(place)*, и его результат передается аргументом в вы-

зываемый метод. Состояние вложенной функции удаляется в методе *destroy* комбинатора аналогично созданию. Таким образом, в указанных случаях в методы вложенной функции передается не весь *place*, полученный методом комбинатора, а возвращаемый функцией *getNestedPlace*, то есть смещенный на *prefix_size*.

4.2 Модификатор DISTINCT

4.2.1 Аргументы

Модификатор *DISTINCT* в качестве аргументов принимает перечисление столбцов, по которым необходимо провести уникализацию. Аргументы, переданные в запросе, принимаются методом *transformArguments* класса *AggregateFunctionCombinatorDistinct*, которая возвращает сформированный из них массив *DataTypes*.

4.2.2 Элементы класса

Для полной функциональности комбинатора необходимы несколько полей класса:

1. *AggregateFunctionPtr nested_func* – вложенная агрегатная функция.
2. *size_t num_arguments* – количество столбцов, по которым необходимо провести агрегацию.
3. *size_t prefix_size* – размер сдвига состояния, необходимый для хранения данных комбинатора.

4.2.3 Инициализация

Конструктор комбинатора принимает в качестве аргументов вложенную агрегатную функцию и набор аргументов функции. В нем происходит инициализация полей *nested_func*, *num_arguments* и *prefix_size* переданной

агрегатной функцией, размером переданных аргументов и размером структуры *AggregateFunctionDistinctData* соответственно.

4.2.4 Данные

Для определения уникальности элемента данных в каждом потоке исполнения все уникальные данные необходимо иметь в общем для всех потоков доступе. Так как для данного модификатора количество повторяющихся элементов данных не имеет значения, а осуществлять проверку на наличие элемента в уже сформированных данных и добавление в них необходимо за константное время, для хранения данных выбрана структура *HashSet*, ранее реализованная в ClickHouse. Данные представлены типом *char**, из-за чего хранить их в хэш-таблице не представляется возможным (равенство значений не означает равенство указателей), поэтому было принято решение хранить преобразованный к *UInt128* хэш, обновляемый принимаемыми данными при помощи метода класса *IColumn*, который предназначен для передачи данных в метод *add*. Для упрощения обращения к структуре хранения данных в методах комбинатора была создана структура *AggregateFunctionDistinctData*, в которой содержится хранилище данных *HashSet<UInt128, ...> set (UInt128* используется как тип хранимых элементов, остальные параметры *HashSet* опустим, так как они не представляют собой важной информации) и метод *bool tryToInsert(... key)*. Внутри этого метода происходит попытка вставки элемента в *set* по переданному ключу *key*, которая возвращает результат поиска и статус вставки (*true*, если она произошла, *false* иначе), а затем из функции возвращается статус вставки.

4.2.5 Метод *add*

Метод *add* принимает несколько важных аргументов: состояние агрегатной функции *place*, набор указателей на колонки *columns*, номер строки *row_num* и *arena* для выделения памяти. В этом методе необходимо установить уникальность получаемого элемента данных и отправить его во вло-

женную агрегатную функцию, если он уникальный. Для этого последовательно выполняется несколько шагов. Вначале создается экземпляр класса *SipHash hash*, затем происходит последовательное обновление хэша методом *updateHashWithValue(row_num, hash)* для каждой колонки в пределах *num_arguments*. После этого *hash* преобразуется в *UInt128 key* с помощью метода *get128* и происходит попытка вставки *data(place).tryToInsert(key)*. В случае успеха (возвращаемое значение = *true*) вызывается метод *add* у вложенной агрегатной функции *nested_func* с переданными в функцию *add* комбинатора параметрами *columns*, *row_num* и *arena*, а также модифицированным *getNestedPlace(place)*.

4.2.6 Метод *merge*

В силу того, что в каждом потоке исполнения данные будут уникальны относительно всех полученных в результате выполнения комбинатора, метод *merge* не нуждается в особенной проработке и состоит из стандартного для комбинаторов вызова метода *merge* вложенной агрегатной функции с переданными в метод комбинатора аргументами и измененным функцией *getNestedPlace* аргументом *place*.

4.3 Модификатор ORDER BY

4.3.1 Изменения в архитектуре

Для того, чтобы уметь разделять все переданные столбцы на те, которые необходимо упорядочить, и те, по которым необходимо упорядочить, нужен параметр, отвечающий за количество столбцов, по которым будет производиться упорядочивание. Он передается в SQL-запросе и должен быть убран в методе *transformParameters*, чтобы не быть переданным во вложенную агрегатную функцию. Аналогично должны быть убраны последние *n* столбцов в методе *transformArguments*, где *n* - это значение переданного параметра, так как во вложенную агрегатную функцию будут передавать-

ся только столбцы, которые необходимо упорядочить в комбинаторе. Но так как изначально метод *transformArguments* принимал только аргументы (то есть столбцы), в нем нельзя было определить количество столбцов, которые необходимо оставить. Таким образом, было принято решение изменить сигнатуру метода *transformArguments*, добавив дополнительный параметр — *const Array&*, отвечающий за параметры. Соответствующие изменения были внесены также в два метода класса *AggregateFunctionFactory*, в которых происходил вызов *transformArguments(arguments)* — теперь в этих местах происходит вызов *transformArguments(arguments, parameters)*. Сигнатура метода *transformArguments* других комбинаторов также была изменена для соответствия интерфейсу.

4.3.2 Аргументы

Модификатор *ORDER BY* в качестве аргументов принимает перечисление всех столбцов, с которыми должен взаимодействовать комбинатор. В методе *transformArguments* класса *AggregateFunctionCombinatorOrderBy* происходит отделение столбцов, которые необходимо упорядочить и передать во вложенную агрегатную функцию, от столбцов, по которым необходимо упорядочивать данные. Это происходит с использованием параметров агрегатной функции — переданный параметр определяет, какое количество столбцов должно быть убрано из аргументов вложенной агрегатной функции. Метод убирает последние *n* столбцов, где *n* — это значение переданного параметра, тем самым определяя, какие именно столбцы будут переданы вложенной агрегатной функции при вызове её методов.

4.3.3 Параметры

Для определения количества упорядочиваемых столбцов в комбинаторе — *OrderBy* передается параметр-число. Так как он нужен исключительно в комбинаторе, в методе *transformParameters* происходит удаление последнего переданного аргумента из списка аргументов, то есть создается массив

параметров для вложенной агрегатной функции без параметра количества столбцов. Вызов функции *groupBy(x ORDER BY y, z)* можно интерпретировать так: *groupByOrderBy(2)(x, y, z)*.

4.3.4 Элементы класса

Для полной функциональности комбинатора необходимы несколько полей класса:

1. *AggregateFunctionPtr nested_func* – вложенная агрегатная функция.
2. *size_t num_arguments* – количество столбцов, по которым необходимо провести агрегацию.
3. *size_t prefix_size* – размер сдвига состояния, необходимый для хранения данных комбинатора.
4. *size_t num_sort_args* – количество столбцов, по которым необходимо провести упорядочивание.

4.3.5 Инициализация

Конструктор комбинатора принимает в качестве аргументов вложенную агрегатную функцию и наборы аргументов и параметров функции. В нем происходит инициализация всех полей класса:

1. *nested_func* – переданной агрегатной функцией.
2. *num_arguments* – размером переданных аргументов.
3. *prefix_size* – размером структуры *AggregateFunctionOrderByData*.
4. *num_sort_args* – значением последнего переданного параметра.

4.3.6 Данные

Для того, чтобы передавать во вложенную агрегатную функцию упорядоченные данные, вначале их необходимо поместить в общее для всех потоков исполнение хранилище при вызове метода *add* комбинатора. Данные было решено хранить в состоянии агрегатной функции, используя структуру *AggregateFunctionOrderByData*. Она хранит в себе массив данных *std :: vector<IColumn*> value*, флаг инициализации *std :: once_flag initialized* и ссылку на арену *Arena * arena*.

4.3.7 Метод *add*

В методе *add* один раз для всех потоков исполнения выполняется инициализация колонок, хранимых в состоянии комбинатора. Одноразовая инициализация достигается с помощью вызова функции *call_once* с флагом *initialized* из состояния комбинатора. Функция заставляет другие потоки исполнения ждать окончания выполнения лямбда-функции первого зашедшего, а затем все потоки пропускают исполнение, так как флаг *initialized* помечается исполненным. Лямбда-функция представляет собой цикл длины *num_arguments*, который инициализирует каждый столбец *data(place).value* пустым значением такого же типа, как *columns* этого же столбца, где *columns* - переданные в метод *add* данные типа *const IColumn***. Далее в цикле длины *num_arguments* в каждый столбец *data(place).value* добавляется значение, лежащее в *columns* этого же столбца на строке *row_num* из аргументов метода. Также в этом методе происходит инициализация арены в состоянии комбинатора аргументом метода: *data(place).arena = arena*. Вызов метода *add* вложенной агрегатной функции не происходит, так как данные, поступающие в нее, должны быть упорядочены.

4.3.8 Метод *insertResultInto*

В методе *insertResultInto* необходимо достать данные из состояния комбинатора, упорядочить по последним *num_sort_args* столбцам и вызвать метод *add* вложенной агрегатной функции для каждого элемента упорядоченных данных. Далее вызвать метод *insertResultInto* у вложенной агрегатной функции. Для того, чтобы извлечь данные из состояния комбинатора, создается массив колонок *const IColumn * **, в который складываются в цикле до *num_arguments* все данные. Далее создается перестановка *IColumn :: Permutation* размера длины данных, и её элементы инициализируются порядковым номером, таким образом создавая нумерацию всех строк данных. Перестановка сортируется с помощью *std :: sort*, значения сравниваются следующим образом: имея два числа *l* и *r* перестановки, в цикле, начиная с *num_sort_args*, сравниваются значения колонки под текущим индексом в строках *l* и *r* с помощью метода *compareAt* колонки. В итоге получается нужным образом упорядоченная перестановка. После этого в цикле до количества строк данных вызывается метод *add* вложенной агрегатной функции со следующими аргументами: как колонки – извлеченные из состояния комбинатора в начале метода данные, как номер строки – значение перестановки для текущего индекса, как арену – сохраненную в состоянии комбинатора арену. После этого вызывается метод *insertResultInto* вложенной агрегатной функции, который помещает только что добавленные методом *add* данные в колонку результата.

4.4 Выводы и результаты

Реализация новых модификаторов схожа со структурой ранее реализованных в ClickHouse комбинаторов, как и предполагалось в главе 3. Было проведено много работы, связанной с определением способа хранения данных в состоянии комбинатора, комбинацией состояний вложенной агрегатной функции и комбинатора в одном состоянии комбинатора, а также уникализацией

и сортировкой данных.

5 Проверка результатов

5.1 Тестирование

Проверка работы модификаторов *DISTINCT* и *ORDER BY* происходила путем сборки проекта ClickHouse с учетом внесенных изменений, запуска сервера и исполнения команд, содержащих агрегатную функцию с суффиксами *-Distinct* и *-OrderBy* на различных наборах данных. Некоторые данные были взяты из примеров данных ClickHouse [16], некоторые выдуманы. Результаты выполнения некоторых команд можно посмотреть в приложениях (разд. 8.3.1). Стоит отметить, что случаев некорректной работы модификаторов отмечено не было.

5.2 Функциональные тесты

5.2.1 Функциональные тесты в ClickHouse

Для тестирования кода в ClickHouse существует несколько видов тестов. Один из них - функциональные тесты. Они пишутся на языке SQL и хранятся по адресу *tests/queries/0_stateless/<test_number>_<test_name>.sql*. К каждому тесту прилагается одноименный файл расширения *.reference*, который генерируется путем запуска теста (*clickhouse-client -n --testmode <test_file>.sql > <test_file>.reference*) и содержит в себе правильные результаты исполнения команд теста. В командах теста можно использовать только встроенные, системные или временные таблицы. В написанных тестах для модификаторов были использованы системные таблицы *system.numbers* и *system.numbers_mt*. Написанные для новых комбинаторов тесты можно посмотреть в приложениях (разд. 8.3.2).

6 Заключение

6.1 Результаты

При выполнении работы было проведено изучение архитектуры агрегатных функций и их комбинаторов, а также некоторых других уже реализованных в ClickHouse структур. Также была полностью продумана архитектура новых комбинаторов, начиная от способа хранения и обработки данных, заканчивая комбинированием состояний комбинатора и вложенной агрегатной функции. В результате работы была расширена функциональность СУБД ClickHouse – добавлены и подключены к фабрике комбинаторов новые модификаторы *DISTINCT* и *ORDER BY*, совместимые со всеми агрегатными функциями, а также были добавлены автоматические тесты, которые покажут, если будут внесены изменения, делающие работу функциональности некорректной. На данный момент оба модификатора находятся в состоянии *review*, то есть еще не находятся в основной кодовой базе проекта, но проходят проверки тестирующей системы и сотрудников ClickHouse. Ссылки на *pull requests*:

1. *DISTINCT* — <https://github.com/ClickHouse/ClickHouse/pull/10930>
2. *ORDER BY* — <https://github.com/ClickHouse/ClickHouse/pull/11234>

6.2 Развитие

В дальнейшем можно внести изменения и улучшить качество кода, если потребуется по итогам *review*, а также внести изменения двух *pull requests* в основную кодовую базу проекта.

7 Источники

1. Кодовая база ClickHouse, GitHub.
<https://github.com/ClickHouse/ClickHouse>

2. Документация по ClickHouse, комбинаторы агрегатных функций.
<https://clickhouse.tech/docs/ru/sql-reference/aggregate-functions-combinators>
3. Документация для разработчиков PostgreSQL, агрегатные функции.
<https://www.postgresql.org/docs/devel/functions-aggregate.html>
4. Кодовая база PostgreSQL, вызов агрегатных функций, FuncCall.
<https://git.postgresql.org/gitweb/?p=postgresql.git;a=blob;f=src/include/nodes/parsenodes.h#l335>
5. Документация для разработчиков MySQL, класс Item_result_field.
https://dev.mysql.com/doc/dev/mysql-server/latest/classItem__result__field.html
6. Документация для разработчиков MySQL, класс Item_sum.
https://dev.mysql.com/doc/dev/mysql-server/latest/classItem__sum.html#aab1364c0f2d629b6fbe83380ecdb3693a5c3953c913c0224cffebbb156524627
7. Документация для разработчиков MySQL, класс Aggregator_distinct.
https://dev.mysql.com/doc/dev/mysql-server/latest/classAggregator__distinct.html
8. Документация для разработчиков MySQL, класс Aggregator.
<https://dev.mysql.com/doc/dev/mysql-server/latest/classAggregator.html>
9. Документация Apache Druid, агрегатные функции.
<https://druid.apache.org/docs/latest/querying/aggregations.html>
10. Кодовая база Druid, агрегатная функция Count.
<https://github.com/apache/druid/blob/master/processing/src/main/java/org/apache/druid/query/aggregation/CountAggregator.java>

11. Документация Apache Druid, модификатор distinct.
<https://druid.apache.org/docs/latest/querying/aggregations.html#approximate-aggregations>
12. Кодовая база Druid, фабрика агрегатных функций HyperUniques.
<https://github.com/apache/druid/blob/master/processing/src/main/java/org/apache/druid/query/aggregation/hyperloglog/HyperUniquesAggregatorFactory.java>
13. Кодовая база Druid, OrderBy.
<https://github.com/apache/druid/blob/master/processing/src/main/java/org/apache/druid/query/groupby/orderby/OrderByColumnSpec.java>
14. Кодовая база MonetDB, операция select.
https://dev.monetdb.org/hg/MonetDB/file/cc19c3e6a79a/sql/server/rel_select.c#l2701
15. Документация по ClickHouse, агрегатные функции.
https://clickhouse.tech/docs/ru/query_language/agg_functions/reference
16. Документация ClickHouse, примеры данных.
<https://clickhouse.tech/docs/ru/getting-started/example-datasets/>

8 Приложения

8.1 Глоссарий

1. Комбинатор - реализация модификатора на языке программирования, позволяющая сочетать себя с агрегатной функцией.
2. Фабрика - структура, написанная на языке программирования, позволяющая по названию создавать нужный экземпляр класса.

- Интерфейс - абстрактный класс, задающий общую структуру классов-наследников.

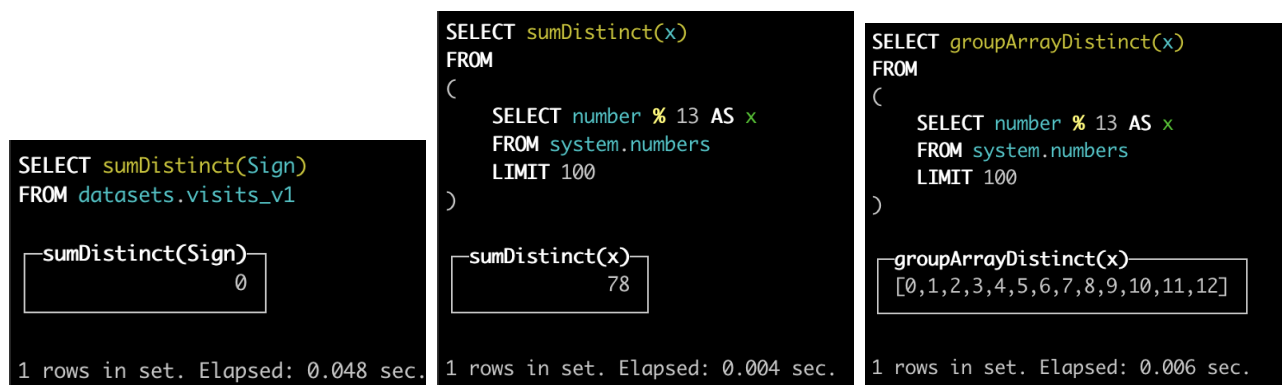
8.2 Список сокращений

- СУБД - система управления базами данных
- БД - база (базы) данных
- SQL - structured query language

8.3 Эксперименты

8.3.1 Ручное тестирование

Результаты исполнения команд в командной строке ClickHouse для тестирования реализации модификатора *DISTINCT* представлены на рис. 9.1. Также видно время исполнения команд – оно достаточно быстрое и соответ-



```
SELECT sumDistinct(Sign)
FROM datasets.visits_v1

sumDistinct(Sign)
0

1 rows in set. Elapsed: 0.048 sec.
```

```
SELECT sumDistinct(x)
FROM (
  SELECT number % 13 AS x
  FROM system.numbers
  LIMIT 100
)

sumDistinct(x)
78

1 rows in set. Elapsed: 0.004 sec.
```

```
SELECT groupArrayDistinct(x)
FROM (
  SELECT number % 13 AS x
  FROM system.numbers
  LIMIT 100
)

groupArrayDistinct(x)
[0,1,2,3,4,5,6,7,8,9,10,11,12]

1 rows in set. Elapsed: 0.006 sec.
```

Рис. 9.1. Примеры исполнения SQL-команд для тестирования *DISTINCT*

ствует стандартному для агрегатной функции времени исполнения. В первом запросе были взяты данные из открытых данных с сайта ClickHouse, столбец *Sign* состоит из значений -1 и 1 , поэтому сумма всех различных равна $-1 + 1 = 0$. Во втором и третьем запросах была взята выборка из системной таблицы *system.numbers*, состоящая из первых 100 элементов (чисел от 0 до 99), взятых по модулю 13. Таким образом различных значений во втором и третьем запросе 13 (это числа от 0 до 12).

Результаты исполнения команд в командной строке ClickHouse для тестирования реализации модификатора *ORDER BY* представлены на рис. 9.2. Дан-

```
SELECT groupArrayOrderBy(1)(x, y)
FROM
(
  SELECT
    number AS x,
    number % 3 AS y
  FROM system.numbers
  LIMIT 10
)
groupArrayOrderBy(1)(x, y)
[0,3,6,9,1,4,7,2,5,8]
1 rows in set. Elapsed: 0.006 sec.
```

```
SELECT groupArrayOrderBy(2)(x, y, z)
FROM
(
  SELECT
    number AS x,
    number % 3 AS y,
    number % 5 AS z
  FROM system.numbers
  LIMIT 10
)
groupArrayOrderBy(2)(x, y, z)
[0,6,3,9,1,7,4,5,2,8]
1 rows in set. Elapsed: 0.005 sec.
```

Рис. 9.2. Примеры исполнения SQL-команд для тестирования *ORDER BY*

ные в этих тестированиях были взяты из системной таблицы *system.numbers* с различными модификациями её столбца. В первом запросе происходит сортировка столбца *x*, состоящего из чисел от 0 до 9, по столбцу *y*, состоящему из значений 0, 1, 2. Таким образом, сначала берутся столбцы, в которых $y = 0$, затем те, в которых $y = 1$ и тд. Аналогичным образом происходит сортировка столбца *x* по столбцам *y* и *z* во втором запросе. Время исполнения также соответствует среднему времени работы агрегатных функций на таком же количестве данных.

8.3.2 Функциональные тесты

Для новых комбинаторов были добавлены функциональные тесты. Набор SQL-команд для тестирования *DISTINCT* находится в папке с тестами в файле *01259_combinator_distinct.sql* (рис. 9.3).

Набор SQL-команд для тестирования *ORDER BY* находится в папке с тестами в файле *01281_combinator_orderby.sql* (рис. 9.4).

<pre>SELECT sum(DISTINCT x) FROM (SELECT number AS x FROM system.numbers LIMIT 1000); SELECT sum(DISTINCT x) FROM (SELECT number % 13 AS x FROM system.numbers LIMIT 1000); SELECT groupArray(DISTINCT x) FROM (SELECT number % 13 AS x FROM system.numbers LIMIT 1000); SELECT groupArray(DISTINCT x) FROM (SELECT number % 13 AS x FROM system.numbers_mt LIMIT 1000); SELECT corrStableDistinct(DISTINCT x, y) FROM (SELECT number % 11 AS x, number % 13 AS y FROM system.numbers LIMIT 1000);</pre>	<pre>499500 78 [0,1,2,3,4,5,6,7,8,9,10,11,12] [0,1,2,3,4,5,6,7,8,9,10,11,12] 5.669227916063075e-17</pre>
--	--

Рис. 9.3. Функциональный тест и ответы к нему для *DISTINCT*

<pre>SELECT groupArrayOrderBy(2)(x, y, z) FROM (SELECT number AS x, number % 3 AS y, number % 5 AS z FROM system.numbers LIMIT 10); SELECT groupArrayOrderBy(1)(x, y) FROM (SELECT number AS x, number % 3 AS y FROM system.numbers_mt LIMIT 10); SELECT groupArrayOrderBy(1)(x, y) FROM (SELECT number AS x, number AS y FROM system.numbers_mt LIMIT 10);</pre>	<pre>[0,6,3,9,1,7,4,5,2,8] [0,3,6,9,1,4,7,2,5,8] [0,1,2,3,4,5,6,7,8,9]</pre>
---	--

Рис. 9.4. Функциональный тест и ответы к нему для *ORDER BY*