**RT.WIDESTORE**

**ИНСТРУКЦИЯ ПО УСТАНОВКЕ**

**ОГЛАВЛЕНИЕ**

[1 НАЗНАЧЕНИЕ ДОКУМЕНТА 4](#_Toc75519588)

[2 ОБЩИЕ ПОЛОЖЕНИЯ 5](#_Toc75519589)

[2.1 Наименование и условное обозначение 5](#_Toc75519590)

[2.2 Область применения 5](#_Toc75519591)

[2.3 Термины, определения и сокращения 5](#_Toc75519592)

[3 НАЗНАЧЕНИЕ СИСТЕМЫ 7](#_Toc75519593)

[3.1 Вид деятельности, для автоматизации которой предназначена Система 7](#_Toc75519594)

[3.2 Преимущества использования столбцой СУБД для обработки аналитических запросов 7](#_Toc75519595)

[4 ОПИСАНИЕ СИСТЕМЫ 10](#_Toc75519596)

[4.1 Системные требования 10](#_Toc75519597)

[4.2 Отличительные особенности СУБД RT.WideStore 10](#_Toc75519598)

[4.2.1 По-настоящему столбцовая СУБД 10](#_Toc75519599)

[4.2.2 Сжатие данных 10](#_Toc75519600)

[4.2.3 Хранение данных на диске 10](#_Toc75519601)

[4.2.4 Параллельная обработка запроса на многих процессорных ядрах 11](#_Toc75519602)

[4.2.5 Распределённая обработка запроса на многих серверах 11](#_Toc75519603)

[4.2.6 Поддержка SQL 11](#_Toc75519604)

[4.2.7 Векторный движок 11](#_Toc75519605)

[4.2.8 Обновление данных в реальном времени 11](#_Toc75519606)

[4.2.9 Наличие индекса 11](#_Toc75519607)

[4.2.10 Подходит для онлайн запросов 11](#_Toc75519608)

[4.2.11 Поддержка приближённых вычислений 11](#_Toc75519609)

[4.2.12 Репликация данных и поддержка целостности 12](#_Toc75519610)

[4.2.13 Особенности, которые могут считаться недостатками 12](#_Toc75519611)

[4.3 Архитектура Системы 12](#_Toc75519612)

[4.3.1 Колонки 13](#_Toc75519613)

[4.3.2 Поля 13](#_Toc75519614)

[4.3.3 Дырявые абстракции (Leaky Abstractions) 13](#_Toc75519615)

[4.3.4 Типы данных (Data Types) 14](#_Toc75519616)

[4.3.5 Блоки (Block) 14](#_Toc75519617)

[4.3.6 Потоки блоков (Block Streams) 15](#_Toc75519618)

[4.3.7 Форматы 16](#_Toc75519619)

[4.3.8 Ввод/вывод 16](#_Toc75519620)

[4.3.9 Таблицы 17](#_Toc75519621)

[4.3.10 Парсеры (Parsers) 18](#_Toc75519622)

[4.3.11 Интерпретаторы 18](#_Toc75519623)

[4.3.12 Функции 18](#_Toc75519624)

[4.3.13 Агрегатные функции 19](#_Toc75519625)

[4.3.14 Сервер 20](#_Toc75519626)

[4.3.15 Выполнение распределенных запросов (Distributed Query Execution) 20](#_Toc75519627)

[4.3.16 Merge Tree 21](#_Toc75519628)

[4.3.17 Репликация 23](#_Toc75519629)

[4.4 Интерфейсы взаимодействия с Системой 24](#_Toc75519630)

[4.4.1 HTTP-интерфейс 24](#_Toc75519631)

[4.4.2 Native TCP 25](#_Toc75519632)

[4.4.3 Консольный клиент 25](#_Toc75519633)

[4.4.4 JDBC-драйвер 25](#_Toc75519634)

[4.4.5 ODBC-драйвер 25](#_Toc75519635)

[4.4.6 C++ клиентская библиотека 25](#_Toc75519636)

[4.4.7 Клиентские библиотеки 26](#_Toc75519637)

[4.4.8 Библиотеки для интеграции 27](#_Toc75519638)

[4.4.9 Визуальные интерфейсы 29](#_Toc75519639)

[5 УСТАНОВКА RT.WIDESTORE 33](#_Toc75519640)

[5.1 Системные требования 33](#_Toc75519641)

[5.2 Доступные варианты установки 33](#_Toc75519642)

[5.2.1 Из DEB пакетов 33](#_Toc75519643)

[5.2.2 Из RPM пакетов 33](#_Toc75519644)

[5.2.3 Из Tgz архивов 34](#_Toc75519645)

[5.2.4 Из Docker образа 34](#_Toc75519646)

[5.2.5 Из исполняемых файлов для нестандартных окружений 34](#_Toc75519647)

[5.2.6 Из исходного кода 35](#_Toc75519648)

[6 ЗАПУСК RT.WIDESTORE 36](#_Toc75519649)

[7 ОБНОВЛЕНИЕ RT.WIDESTORE 37](#_Toc75519650)

# НАЗНАЧЕНИЕ ДОКУМЕНТА

Настоящий документ включает вводную часть, описывающую систему «RT.WideStore», ее возможности и отличительные особенности, а также подробную информацию об установке, запуске и обновлению «RT.WideStore».

# ОБЩИЕ ПОЛОЖЕНИЯ

## Наименование и условное обозначение

Наименование системы: RT.WideStore.

## Область применения

RT.WideStore – столбцовая система управления базами данных (СУБД), предназначенная для онлайн обработки аналитических запросов (OLAP).

## Термины, определения и сокращения

В настоящем документе использованы и определены следующие термины и сокращения:

| Термин/Сокращение | Определение |
| --- | --- |
| Apache ZooKeeper | Программный проект Apache Software Foundation. По сути, это сервис для распределенных систем, предлагающий иерархическое хранилище ключей и значений, которое используется для предоставления распределенных:   * службы конфигурации, * службы синхронизации, * реестра имен,   для больших распределенных систем. |
| CPU | Процессор |
| [GDPR](https://gdpr-info.eu/) | (англ. General Data Protection Regulation) общий регламент по защите данных жителей ЕС, вступил в действие с 25 мая 2018. |
| ODBC | (англ. Open Database Connectivity) – широко распространенный интерфейс прикладного программирования (API) для доступа к базе данных. Он основан на спецификациях интерфейса уровня вызовов (CLI) от Open Group и ISO / IEC для API баз данных и использует язык структурированных запросов (SQL) в качестве языка доступа к базе данных. |
| OLAP | (англ. online analytical processing, интерактивная аналитическая обработка) – технология обработки данных, заключающаяся в подготовке суммарной (агрегированной) информации на основе больших массивов данных, структурированных по многомерному принципу. |
| SQL | (англ. structured query language, язык структурированных запросов) –декларативный язык программирования, применяемый для создания, модификации и управления данными в реляционной базе данных, управляемой соответствующей [системой управления базами данных](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D1%83%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F_%D0%B1%D0%B0%D0%B7%D0%B0%D0%BC%D0%B8_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85). |
| SSD | (англ. Solid-State Drive,твердотельный накопитель) – компьютерное энергонезависимое [немеханическое](https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%85%D0%B0%D0%BD%D0%B8%D0%B7%D0%BC) [запоминающее устройство](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%BF%D0%BE%D0%BC%D0%B8%D0%BD%D0%B0%D1%8E%D1%89%D0%B5%D0%B5_%D1%83%D1%81%D1%82%D1%80%D0%BE%D0%B9%D1%81%D1%82%D0%B2%D0%BE) на основе [микросхем памяти](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C), альтернатива [HDD](https://ru.wikipedia.org/wiki/%D0%96%D1%91%D1%81%D1%82%D0%BA%D0%B8%D0%B9_%D0%B4%D0%B8%D1%81%D0%BA). |
| SSE 4.2 | Один из наборов инструкций процессора. |
| URL | ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) Uniform Resource Locator, унифицированный указатель ресурса) – система унифицированных адресов электронных ресурсов, или единообразный определитель местонахождения ресурса ([файла](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB)).  Используется как стандарт записи ссылок на объекты в [Интернете](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D0%BD%D0%B5%D1%82). |
| Система | Система «RT.WideStore». |
| СУБД | Система управления базами данных. |

# НАЗНАЧЕНИЕ СИСТЕМЫ

## Вид деятельности, для автоматизации которой предназначена Система

СУБД RT.WideStore из-за своей архитектуры оптимальна для выполнения онлайн запросов, используемых для построения отчетов на основе больших массивов данных, структурированных по многомерному принципу.

## Преимущества использования столбцой СУБД для обработки аналитических запросов

RT.WideStore представляет собой столбцовую система управления базами данных.

В обычной, «строковой» СУБД, данные относящиеся к одной строке физически хранятся рядом.

Примеры строковых СУБД: MySQL, Postgres, MS SQL Server.

В «столбцовых» СУБД, значения из разных столбцов хранятся отдельно, а данные одного столбца - вместе.

Примеры столбцовых СУБД: Vertica, Paraccel (Actian Matrix, Amazon Redshift), Sybase IQ, Exasol, Infobright, InfiniDB, MonetDB (VectorWise, Actian Vector), LucidDB, SAP HANA, Google Dremel, Google PowerDrill, Druid, kdb+.

Разный порядок хранения данных лучше подходит для разных сценариев работы.

Сценарий работы с данными – это то, какие производятся запросы, как часто и в каком соотношении; сколько читается данных на запросы каждого вида – строк, столбцов, байт; как соотносятся чтения и обновления данных; какой рабочий размер данных и насколько локально он используется; используются ли транзакции и с какой изолированностью; какие требования к дублированию данных и логической целостности; требования к задержкам на выполнение и пропускной способности запросов каждого вида и т. п.

Чем больше нагрузка на систему, тем более важной становится специализация под сценарий работы, и тем более конкретной становится эта специализация. Не существует системы, одинаково хорошо подходящей под существенно различные сценарии работы. Если система подходит под широкое множество сценариев работы, то при достаточно большой нагрузке, система будет справляться со всеми сценариями работы плохо, или справляться хорошо только с одним из сценариев работы.

Ключевые особенности OLAP сценария работы:

* подавляющее большинство запросов – на чтение;
* данные обновляются достаточно большими пачками (> 1000 строк), а не по одной строке, или не обновляются вообще;
* данные добавляются в БД, но не изменяются;
* при чтении, вынимается достаточно большое количество строк из БД, но только небольшое подмножество столбцов;
* таблицы являются «широкими», то есть, содержат большое количество столбцов;
* запросы идут сравнительно редко (обычно не более сотни в секунду на сервер);
* при выполнении простых запросов, допустимы задержки в районе 50 мс;
* значения в столбцах достаточно мелкие - числа и небольшие строки (пример - 60 байт на URL);
* требуется высокая пропускная способность при обработке одного запроса (до миллиардов строк в секунду на один сервер);
* транзакции отсутствуют;
* низкие требования к консистентности данных;
* в запросе одна большая таблица, все таблицы кроме одной маленькие;
* результат выполнения запроса существенно меньше исходных данных - то есть, данные фильтруются или агрегируются; результат выполнения помещается в оперативку на одном сервере.

Легко видеть, что OLAP сценарий работы существенно отличается от других распространённых сценариев работы (например, OLTP или Key-Value сценариев работы). Таким образом, не имеет никакого смысла пытаться использовать OLTP или Key-Value БД для обработки аналитических запросов, если вы хотите получить высокую производительность. Например, если вы попытаетесь использовать для аналитики MongoDB или Redis - вы получите низкую производительность по сравнению с OLAP-СУБД.

Столбцовые СУБД лучше (от 100 раз по скорости обработки большинства запросов) подходят для OLAP сценария работы.

Причины, по которым столбцовые СУБД лучше подходят для OLAP сценария:

1. По вводу-выводу:
2. Для выполнения аналитического запроса, требуется прочитать небольшое количество столбцов таблицы. В столбцовой БД для этого можно читать только нужные данные. Например, если вам требуется только 5 столбцов из 100, то следует рассчитывать на 20-кратное уменьшение ввода-вывода.
3. Так как данные читаются пачками, то их проще сжимать. Данные, лежащие по столбцам также лучше сжимаются. За счёт этого, дополнительно уменьшается объём ввода-вывода.
4. За счёт уменьшения ввода-вывода, больше данных влезает в системный кэш.

Например, для запроса «посчитать количество записей для каждой рекламной системы», требуется прочитать один столбец «идентификатор рекламной системы», который занимает 1 байт в несжатом виде. Если большинство переходов было не с рекламных систем, то можно рассчитывать хотя бы на десятикратное сжатие этого столбца. При использовании быстрого алгоритма сжатия, возможно разжатие данных со скоростью более нескольких гигабайт несжатых данных в секунду. То есть, такой запрос может выполняться со скоростью около нескольких миллиардов строк в секунду на одном сервере. На практике, такая скорость действительно достигается.

1. По вычислениям:

Так как для выполнения запроса надо обработать достаточно большое количество строк, становится актуальным диспетчеризовывать все операции не для отдельных строк, а для целых векторов, или реализовать движок выполнения запроса так, чтобы издержки на диспетчеризацию были примерно нулевыми. Если этого не делать, то при любой не слишком плохой дисковой подсистеме, интерпретатор запроса неизбежно упрётся в CPU.

Имеет смысл не только хранить данные по столбцам, но и обрабатывать их, по возможности, тоже по столбцам.

Есть два способа это сделать:

1. Векторный движок. Все операции пишутся не для отдельных значений, а для векторов. То есть, вызывать операции надо достаточно редко, и издержки на диспетчеризацию становятся пренебрежимо маленькими. Код операции содержит в себе хорошо оптимизированный внутренний цикл.
2. Кодогенерация. Для запроса генерируется код, в котором подставлены все косвенные вызовы.

В «обычных» БД этого не делается, так как не имеет смысла при выполнении простых запросов. Хотя есть исключения. Например, в MemSQL кодогенерация используется для уменьшения latency при выполнении SQL запросов. Для сравнения, в аналитических СУБД требуется оптимизация throughput, а не latency.

Стоит заметить, что для эффективности по CPU требуется, чтобы язык запросов был декларативным (SQL, MDX) или хотя бы векторным (J, K). То есть, чтобы запрос содержал циклы только в неявном виде, открывая возможности для оптимизации.

# ОПИСАНИЕ СИСТЕМЫ

## Системные требования

RT.WideStore может работать на любой операционной системе Linux, FreeBSD или Mac OS X с архитектурой процессора x86\_64, AArch64 или PowerPC64LE.

## Отличительные особенности СУБД RT.WideStore

### По-настоящему столбцовая СУБД

В по-настоящему столбцовой СУБД рядом со значениями не хранится никаких лишних данных. Например, должны поддерживаться значения постоянной длины, чтобы не хранить рядом со значениями типа «число» их длины. Для примера, миллиард значений типа UInt8 (целое число, занимающее 8 байт) должен действительно занимать в несжатом виде около 1GB, иначе это сильно ударит по эффективности использования CPU. Очень важно хранить данные компактно (без «мусора») в том числе в несжатом виде, так как скорость разжатия (использование CPU) зависит, в основном, от объёма несжатых данных.

Этот пункт пришлось выделить, так как существуют системы, которые могут хранить значения отдельных столбцов по отдельности, но не могут эффективно выполнять аналитические запросы в силу оптимизации под другой сценарий работы. Примеры: HBase, BigTable, Cassandra, HyperTable. В этих системах вы получите пропускную способность в районе сотен тысяч строк в секунду, но не сотен миллионов строк в секунду.

Также стоит заметить, что RT.WideStore является системой управления базами данных, а не одной базой данных. То есть, RT.WideStore позволяет создавать таблицы и базы данных в runtime, загружать данные и выполнять запросы без переконфигурирования и перезапуска сервера.

### Сжатие данных

Некоторые столбцовые СУБД (InfiniDB CE, MonetDB) не используют сжатие данных. Однако сжатие данных действительно играет одну из ключевых ролей в демонстрации отличной производительности. RT.WideStore может использовать сжатие данных.

### Хранение данных на диске

Многие столбцовые СУБД (SAP HANA, Google PowerDrill) могут работать только в оперативной памяти. Такой подход стимулирует выделять больший бюджет на оборудование, чем фактически требуется для анализа в реальном времени. RT.WideStore спроектирован для работы на обычных жестких дисках, что обеспечивает низкую стоимость хранения на гигабайт данных, но SSD и дополнительная оперативная память тоже полноценно используются, если доступны.

### Параллельная обработка запроса на многих процессорных ядрах

Большие запросы естественным образом распараллеливаются, используя все необходимые ресурсы из доступных на сервере.

### Распределённая обработка запроса на многих серверах

Почти все перечисленные ранее столбцовые СУБД не поддерживают распределённую обработку запроса.

В RT.WideStore данные могут быть расположены на разных шардах. Каждый шард может представлять собой группу реплик, которые используются для отказоустойчивости. Запрос будет выполнен на всех шардах параллельно. Это делается прозрачно для пользователя.

### Поддержка SQL

RT.WideStore поддерживает декларативный язык запросов на основе SQL и во многих случаях совпадающий с SQL стандартом.

Поддерживаются GROUP BY, ORDER BY, подзапросы в секциях FROM, IN, JOIN, а также скалярные подзапросы.

Зависимые подзапросы и оконные функции не поддерживаются.

### Векторный движок

Данные не только хранятся по столбцам, но и обрабатываются по векторам – кусочкам столбцов. За счёт этого достигается высокая эффективность по CPU.

### Обновление данных в реальном времени

RT.WideStore поддерживает таблицы с первичным ключом. Для того, чтобы можно было быстро выполнять запросы по диапазону первичного ключа, данные инкрементально сортируются с помощью merge дерева. За счёт этого, поддерживается постоянное добавление данных в таблицу. Блокировки при добавлении данных отсутствуют.

### Наличие индекса

Физическая сортировка данных по первичному ключу позволяет получать данные для конкретных его значений или их диапазонов с низкими задержками - менее десятков миллисекунд.

### Подходит для онлайн запросов

Низкие задержки позволяют не откладывать выполнение запроса и не подготавливать ответ заранее, а выполнять его именно в момент загрузки страницы пользовательского интерфейса. То есть, в режиме онлайн.

### Поддержка приближённых вычислений

RT.WideStore предоставляет различные способы разменять точность вычислений на производительность:

1. Система содержит агрегатные функции для приближённого вычисления количества различных значений, медианы и квантилей.
2. Поддерживается возможность выполнить запрос на основе части (выборки) данных и получить приближённый результат. При этом, с диска будет считано пропорционально меньше данных.
3. Поддерживается возможность выполнить агрегацию не для всех ключей, а для ограниченного количества первых попавшихся ключей. При выполнении некоторых условий на распределение ключей в данных, это позволяет получить достаточно точный результат с использованием меньшего количества ресурсов.

### Репликация данных и поддержка целостности

Используется асинхронная multimaster репликация. После записи на любую доступную реплику, данные распространяются на все остальные реплики в фоне. Система поддерживает полную идентичность данных на разных репликах. Восстановление после большинства сбоев осуществляется автоматически, а в сложных случаях — полуавтоматически. При необходимости, можно [включить кворумную запись](https://clickhouse.tech/docs/ru/operations/settings/settings/) данных.

### Особенности, которые могут считаться недостатками

1. Отсутствие полноценных транзакций.
2. Возможность изменять или удалять ранее записанные данные с низкими задержками и высокой частотой запросов не предоставляется. Есть массовое удаление и изменение данных для очистки более не нужного или соответствия [GDPR](https://gdpr-info.eu/).
3. Разреженный индекс делает RT.WideStore плохо пригодным для точечных чтений одиночных строк по своим ключам.

## Архитектура Системы

RT.WideStore - полноценная колоночная СУБД. Данные хранятся в колонках, а в процессе обработки - в массивах (векторах или фрагментах (chunk’ах) колонок). По возможности операции выполняются на массивах, а не на индивидуальных значениях. Это называется “векторизованное выполнения запросов” (vectorized query execution), и помогает снизить стоимость фактической обработки данных.

Эта идея не нова. Такой подход использовался в APL (A programming language, 1957) и его потомках: A + (диалект APL), J (1990), K (1993) и Q (язык программирования Kx Systems, 2003). Программирование на массивах (Array programming) используется в научных вычислительных системах. Эта идея не является чем-то новым и для реляционных баз данных: например, она используется в системе VectorWise (так же известной как Actian Vector Analytic Database от Actian Corporation).

Существует два различных подхода для увеличения скорости обработки запросов: выполнение векторизованного запроса и генерация кода во время выполнения (runtime code generation). В последнем случае код генерируется на лету для каждого типа запроса, удаляя все косвенные и динамические обращения. Ни один из этих подходов не имеет явного преимущества. Генерация кода во время выполнения выигрывает, если объединяет большое число операций, таким образом полностью используя вычислительные блоки и конвейер CPU. Выполнение векторизованного запроса может быть менее практично потому, что задействует временные векторы данных, которые должны быть записаны и прочитаны из кэша. Если временные данные не помещаются в L2 кэш, будут проблемы. С другой стороны, выполнение векторизованного запроса упрощает использование SIMD инструкций CPU. [Научная работа](http://15721.courses.cs.cmu.edu/spring2016/papers/p5-sompolski.pdf) наших друзей показывает преимущества сочетания обоих подходов. RT.WideStore использует выполнение векторизованного запроса и имеет ограниченную начальную поддержку генерации кода во время выполнения.

### Колонки

Для представления столбцов в памяти (фактически, фрагментов столбцов) используется интерфейс IColumn. Интерфейс предоставляет вспомогательные методы для реализации различных реляционных операторов. Почти все операции иммутабельные: они не изменяют оригинальных колонок, а создают новую с измененными значениями. Например, метод "IColumn :: filter" принимает фильтр – набор байт. Он используется для реляционных операторов WHERE и HAVING. Другой пример: метод "IColumn :: permute" используется для поддержки ORDER BY, метод "IColumn :: cut" - LIMIT и т. д.

Различные реализации IColumn (ColumnUInt8, ColumnString и т. д.) отвечают за распределение данных колонки в памяти. Для колонок целочисленного типа это один смежный массив, такой как std :: vector. Для колонок типа String и Array это два вектора: один для всех элементов массивов, располагающихся смежно, второй для хранения смещения до начала каждого массива. Также существует реализация ColumnConst, в которой хранится только одно значение в памяти, но выглядит как колонка.

### Поля

Тем не менее, можно работать и с индивидуальными значениями. Для представления индивидуальных значений используется Поле (Field). Field – размеченное объединение UInt64, Int64, Float64, String и Array. IColumn имеет метод оператор [] для получения значения по индексу n как Field, а также метод insert для добавления Field в конец колонки. Эти методы не очень эффективны, так как требуют временных объектов Field, представляющих индивидуальное значение. Есть более эффективные методы, такие как insertFrom, insertRangeFrom и т.д.

Field не несет в себе достаточно информации о конкретном типе данных в таблице. Например, UInt8, UInt16, UInt32 и UInt64 в Field представлены как UInt64.

### Дырявые абстракции (Leaky Abstractions)

IColumn предоставляет методы для общих реляционных преобразований данных, но они не отвечают всем потребностям. Например, ColumnUInt64 не имеет метода для вычисления суммы двух столбцов, а ColumnString не имеет метода для запуска поиска по подстроке. Эти бесчисленные процедуры реализованы вне IColumn.

Различные функции на колонках могут быть реализованы обобщенным, неэффективным путем, используя IColumn методы для извлечения значений Field, или специальным путем, используя знания о внутреннем распределение данных в памяти в конкретной реализации IColumn. Для этого функции приводятся к конкретному типу IColumn и работают напрямую с его внутренним представлением. Например, в ColumnUInt64 есть метод getData, который возвращает ссылку на внутренний массив, чтение и заполнение которого, выполняется отдельной процедурой напрямую. Фактически, мы имеем "дырявую абстракции", обеспечивающие эффективные специализации различных процедур.

### Типы данных (Data Types)

IDataType отвечает за сериализацию и десериализацию - чтение и запись фрагментов колонок или индивидуальных значений в бинарном или текстовом формате.

IDataType точно соответствует типам данных в таблицах - DataTypeUInt32, DataTypeDateTime, DataTypeString и т. д.

IDataType и IColumn слабо связаны друг с другом. Различные типы данных могут быть представлены в памяти с помощью одной реализации IColumn. Например, и DataTypeUInt32, и DataTypeDateTime в памяти представлены как ColumnUInt32 или ColumnConstUInt32. В добавок к этому, один тип данных может быть представлен различными реализациями IColumn. Например, DataTypeUInt8 может быть представлен как ColumnUInt8 и ColumnConstUInt8.

IDataType хранит только метаданные. Например, DataTypeUInt8 не хранить ничего (кроме скрытого указателя vptr), а DataTypeFixedString хранит только N (фиксированный размер строки).

В IDataType есть вспомогательные методы для данных различного формата. Среди них методы сериализации значений, допускающих использование кавычек, сериализации значения в JSON или XML. Среди них нет прямого соответствия форматам данных. Например, различные форматы Pretty и TabSeparated могут использовать один вспомогательный метод serializeTextEscaped интерфейса IDataType.

### Блоки (Block)

Block это контейнер, который представляет фрагмент (chunk) таблицы в памяти. Это набор троек – (IColumn, IDataType, имя колонки). В процессе выполнения запроса, данные обрабатываются Blockами. Если у нас есть Block, значит у нас есть данные (в объекте IColumn), информация о типе (в IDataType), которая говорит нам, как работать с колонкой, и имя колонки (оригинальное имя колонки таблицы или служебное имя, присвоенное для получения промежуточных результатов вычислений).

При вычислении некоторой функции на колонках в блоке мы добавляем еще одну колонку с результатами в блок, не трогая колонки аргументов функции, потому что операции иммутабельные. Позже ненужные колонки могут быть удалены из блока, но не модифицированы. Это удобно для устранения общих подвыражений.

Блоки создаются для всех обработанных фрагментов данных. Напоминаем, что одни и те же типы вычислений, имена колонок и типы переиспользуются в разных блоках и только данные колонок изменяются. Лучше разделить данные и заголовок блока потому, что в блоках маленького размера мы имеем большой оверхэд по временным строкам при копировании умных указателей (shared\_ptrs) и имен колонок.

### Потоки блоков (Block Streams)

Потоки блоков обрабатывают данные. Мы используем потоки блоков для чтения данных, трансформации или записи данных куда-либо. IBlockInputStream предоставляет метод read для получения следующего блока, пока это возможно, и метод write, чтобы продвигать (push) блок куда-либо.

Потоки отвечают за:

1. Чтение и запись в таблицу. Таблица лишь возвращает поток для чтения или записи блоков.
2. Реализацию форматов данных. Например, при выводе данных в терминал в формате Pretty, вы создаете выходной поток блоков, который форматирует поступающие в него блоки.
3. Трансформацию данных. Допустим, у вас есть IBlockInputStream и вы хотите создать отфильтрованный поток. Вы создаете FilterBlockInputStream и инициализируете его вашим потоком. Затем вы тянете (pull) блоки из FilterBlockInputStream, а он тянет блоки исходного потока, фильтрует их и возвращает отфильтрованные блоки вам. Таким образом построены конвейеры выполнения запросов.

Имеются и более сложные трансформации. Например, когда вы тянете блоки из AggregatingBlockInputStream, он считывает все данные из своего источника, агрегирует их, и возвращает поток агрегированных данных вам. Другой пример: конструктор UnionBlockInputStream принимает множество источников входных данных и число потоков. Такой Stream работает в несколько потоков и читает данные источников параллельно.

Потоки блоков используют «втягивающий» (pull) подход к управлению потоком выполнения: когда вы вытягиваете блок из первого потока, он, следовательно, вытягивает необходимые блоки из вложенных потоков, так и работает весь конвейер выполнения. Ни «pull» ни «push» не имеют явного преимущества, потому что поток управления неявный, и это ограничивает в реализации различных функций, таких как одновременное выполнение нескольких запросов (слияние нескольких конвейеров вместе). Это ограничение можно преодолеть с помощью сопрограмм (coroutines) или просто запуском дополнительных потоков, которые ждут друг друга. У нас может быть больше возможностей, если мы сделаем поток управления явным: если мы локализуем логику для передачи данных из одной расчетной единицы в другую вне этих расчетных единиц.

Следует отметить, что конвейер выполнения запроса создает временные данные на каждом шаге. Мы стараемся сохранить размер блока достаточно маленьким, чтобы временные данные помещались в кэш процессора. При таком допущении запись и чтение временных данных практически бесплатны по сравнению с другими расчетами. Мы могли бы рассмотреть альтернативу, которая заключается в том, чтобы объединить многие операции в конвеере вместе. Это может сделать конвейер как можно короче и удалить большую часть временных данных, что может быть преимуществом, но у такого подхода также есть недостатки. Например, разделенный конвейер позволяет легко реализовать кэширование промежуточных данных, использование промежуточных данных из аналогичных запросов, выполняемых одновременно, и объединение конвейеров для аналогичных запросов.

### Форматы

Форматы данных реализуются с помощью потоков блоков. Есть форматы представления (presentational), пригодные только для вывода данных клиенту, такие как Pretty формат, который предоставляет только IBlockOutputStream. И есть форматы ввода/вывода, такие как TabSeparated или JSONEachRow.

Существуют также потоки строк: IRowInputStream и IRowOutputStream. Они позволяют вытягивать/выталкивать данные отдельными строками, а не блоками. Они нужны только для упрощения реализации ориентированных на строки форматов. Обертка BlockInputStreamFromRowInputStream и BlockOutputStreamFromRowOutputStream позволяет конвертировать потоки, ориентированные на строки, в обычные потоки, ориентированные на блоки.

### Ввод/вывод

Для байт-ориентированных ввода/вывода существуют абстрактные классы ReadBuffer и WriteBuffer. Они используются вместо C++ iostream. Не волнуйтесь: каждый зрелый проект C++ использует что-то другое вместо iostream по уважительным причинам.

ReadBuffer и WriteBuffer это просто непрерывный буфер и курсор, указывающий на позицию в этом буфере. Реализации могут как владеть, так и не владеть памятью буфера. Существует виртуальный метод заполнения буфера следующими данными (для ReadBuffer) или сброса буфера куда-нибудь (например, WriteBuffer). Виртуальные методы редко вызываются.

Реализации ReadBuffer/WriteBuffer используются для работы с файлами и файловыми дескрипторами, а также сетевыми сокетами, для реализации сжатия (CompressedWriteBuffer инициализируется вместе с другим WriteBuffer и осуществляет сжатие данных перед записью в него), и для других целей – названия ConcatReadBuffer, LimitReadBuffer, и HashingWriteBuffer говорят сами за себя.

Буферы чтения/записи имеют дело только с байтами. В заголовочных файлах ReadHelpers и WriteHelpers объявлены некоторые функции, чтобы помочь с форматированием ввода/вывода. Например, есть помощники для записи числа в десятичном формате.

Давайте посмотрим, что происходит, когда вы хотите вывести результат в JSON формате в стандартный вывод (stdout). У вас есть результирующий набор данных, готовый к извлечению из IBlockInputStream. Вы создаете WriteBufferFromFileDescriptor(STDOUT\_FILENO) чтобы записать байты в stdout. Вы создаете JSONRowOutputStream, инициализируете с этим WriteBuffer'ом, чтобы записать строки JSON в stdout. Кроме того, вы создаете BlockOutputStreamFromRowOutputStream, реализуя IBlockOutputStream. Затем вызывается copyData для передачи данных из IBlockInputStream в IBlockOutputStream и все работает. Внутренний JSONRowOutputStream будет писать в формате JSON различные разделители и вызвать IDataType::serializeTextJSON метод со ссылкой на IColumn и номер строки в качестве аргументов. Следовательно, IDataType::serializeTextJSON вызовет метод из WriteHelpers.h: например, writeText для числовых типов и writeJSONString для DataTypeString.

### Таблицы

Интерфейс IStorage служит для отображения таблицы. Различные движки таблиц являются реализациями этого интерфейса. Примеры StorageMergeTree, StorageMemory и так далее. Экземпляры этих классов являются просто таблицами.

Ключевые методы IStorage это read и write. Есть и другие варианты – alter, rename, drop и так далее. Метод read принимает следующие аргументы: набор столбцов для чтения из таблицы, AST запрос и желаемое количество потоков для вывода. Он возвращает один или несколько объектов IBlockInputStream и информацию о стадии обработки данных, которая была завершена внутри табличного движка во время выполнения запроса.

В большинстве случаев метод read отвечает только за чтение указанных столбцов из таблицы, а не за дальнейшую обработку данных. Вся дальнейшая обработка данных осуществляется интерпретатором запросов и не входит в сферу ответственности IStorage.

Но есть и заметные исключения:

* AST запрос, передающийся в метод read, может использоваться движком таблицы для получения информации о возможности использования индекса и считывания меньшего количества данных из таблицы.
* Иногда движок таблиц может сам обрабатывать данные до определенного этапа. Например, StorageDistributed можно отправить запрос на удаленные серверы, попросить их обработать данные до этапа, когда данные с разных удаленных серверов могут быть объединены, и вернуть эти предварительно обработанные данные. Затем интерпретатор запросов завершает обработку данных.

Метод read может возвращать несколько объектов IBlockInputStream, позволяя осуществлять параллельную обработку данных. Эти несколько блочных входных потоков могут считываться из таблицы параллельно. Затем вы можете обернуть эти потоки различными преобразованиями (такими как вычисление выражений или фильтрация), которые могут быть вычислены независимо, и создать UnionBlockInputStream поверх них, чтобы читать из нескольких потоков параллельно.

Есть и другие варианты. Например, TableFunction возвращает временный объект IStorage, который можно подставить во FROM.

Чтобы получить быстрое представление о том, как реализовать свой движок таблиц, посмотрите на что-то простое, например, StorageMemory или StorageTinyLog.

В качестве результата выполнения метода read, IStorage возвращает QueryProcessingStage – информацию о том, какие части запроса были обработаны внутри хранилища.

### Парсеры (Parsers)

Написанный от руки парсер, анализирующий запрос, работает по методу рекурсивного спуска. Например, ParserSelectQuery просто рекурсивно вызывает нижестоящие парсеры для различных частей запроса. Парсеры создают абстрактное синтаксическое дерево (AST). AST представлен узлами, которые являются экземплярами IAST.

Генераторы парсеров не используются по историческим причинам.

### Интерпретаторы

Интерпретаторы отвечают за создание конвейера выполнения запроса из AST. Есть простые интерпретаторы, такие как InterpreterExistsQuery и InterpreterDropQuery или более сложный InterpreterSelectQuery. Конвейер выполнения запроса представляет собой комбинацию входных и выходных потоков блоков. Например, результатом интерпретации SELECT запроса является IBlockInputStream для чтения результирующего набора данных; результат интерпретации INSERT запроса – это IBlockOutputStream, для записи данных, предназначенных для вставки; результат интерпретации INSERT SELECT запроса – это IBlockInputStream, который возвращает пустой результирующий набор при первом чтении, но копирует данные из SELECT к INSERT.

InterpreterSelectQuery использует ExpressionAnalyzer и ExpressionActions механизмы для анализа запросов и преобразований. Именно здесь выполняется большинство оптимизаций запросов на основе правил. ExpressionAnalyzer написан довольно грязно и должен быть переписан: различные преобразования запросов и оптимизации должны быть извлечены в отдельные классы, чтобы позволить модульные преобразования или запросы.

### Функции

Существуют обычные функции и агрегатные функции. Агрегатные функции смотрите в следующем разделе.

Обычный функции не изменяют число строк и работают так, как если бы обрабатывали каждую строку независимо. В действительности же, функции вызываются не к отдельным строкам, а блокам данных для реализации векторизованного выполнения запросов.

Некоторые функции, такие как [blockSize](https://clickhouse.tech/docs/ru/sql-reference/functions/other-functions/#function-blocksize), [rowNumberInBlock](https://clickhouse.tech/docs/ru/sql-reference/functions/other-functions/#function-rownumberinblock), и [runningAccumulate](https://clickhouse.tech/docs/ru/sql-reference/functions/other-functions/#runningaccumulate), эксплуатируют блочную обработку и нарушают независимость строк.

RT.WideStore имеет сильную типизацию, поэтому нет никакого неявного преобразования типов. Если функция не поддерживает определенную комбинацию типов, она создает исключение. Но функции могут работать (перегружаться) для многих различных комбинаций типов. Например, функция plus (для реализации + оператор) работает для любой комбинации числовых типов: UInt8 + Float32, UInt16 + Int8 и так далее. Кроме того, некоторые вариадические функции, такие как concat, могут принимать любое количество аргументов.

Реализация функции может быть немного неудобной, поскольку функция явно определяет поддерживаемые типы данных и поддерживается IColumns. Например, в plus функция имеет код, генерируемый экземпляром шаблона C++ для каждой комбинации числовых типов, а также постоянные или непостоянные левые и правые аргументы.

Это отличное место для реализации генерации кода во время выполнения, чтобы избежать раздувания кода шаблона. Кроме того, он позволяет добавлять слитые функции, такие как fused multiply-add или выполнять несколько сравнений в одной итерации цикла.

Из-за векторизованного выполнения запроса функции не закорачиваются. Например, если вы пишете WHERE f(x) AND g(y), обе части вычисляются, даже для строк, когда f(x) равно нулю (за исключением тех случаев, когда f(x) является нулевым постоянным выражением). Но если избирательность условия f(x) высока, и расчет f(x) обходится гораздо дешевле, чем g(y), лучше всего разделить вычисление на этапы. На первом этапе вычислить f(x), отфильтровать результирующие столбцы, а затем вычислять g(y) только для меньших, отфильтрованных фрагментов данных.

### Агрегатные функции

Агрегатные функции – это функции с состоянием (stateful). Они накапливают переданные значения в некотором состоянии и позволяют получать результаты из этого состояния. Работа с ними осуществляется с помощью интерфейса IAggregateFunction. Состояния могут быть как простыми (состояние для AggregateFunctionCount это всего лишь одна переменная типа UInt64) так и довольно сложными (состояние AggregateFunctionUniqCombined представляет собой комбинацию линейного массива, хэш-таблицы и вероятностной структуры данных HyperLogLog).

Состояния распределяются в Arena (пул памяти) для работы с несколькими состояниями при выполнении запроса GROUP BY высокой кардинальности (большим числом уникальных данных). Состояния могут иметь нетривиальный конструктор и деструктор: например, сложные агрегатные состояния могут сами аллоцировать дополнительную память. Потому к созданию и уничтожению состояний, правильной передаче владения и порядку уничтожения следует уделять больше внимание.

Агрегатные состояния могут быть сериализованы и десериализованы для передачи их по сети во время выполнения распределенного запроса или для записи их на диск при дефиците оперативной памяти. Они даже могут храниться в таблице с DataTypeAggregateFunction, чтобы позволяет выполнять инкрементное агрегирование данных.

Формат сериализации данных для состояний агрегатных функций в настоящее время не версионируется. Это нормально, если агрегатные состояния хранятся только временно. Но у нас есть такая возможность AggregatingMergeTree механизм таблиц для инкрементной агрегации, и люди уже используют его в эксплуатации. Именно по этой причине требуется помнить об обратная совместимости при изменении формата сериализации для любой агрегатной функции.

### Сервер

Сервер предоставляет несколько различных интерфейсов.

* HTTP интерфейс для любых сторонних клиентов.
* TCP интерфейс для родного RT.WideStore клиента и межсерверной взаимодействия при выполнении распределенных запросов.
* Интерфейс для передачи данных при репликации.

Внутри простой многопоточный сервер без корутин (coroutines), файберов (fibers) и т.д. Поскольку сервер не предназначен для обработки большого количества простых запросов, а ориентирован на обработку сложных запросов относительно низкой интенсивности, каждый из потоков может обрабатывать огромное количество аналитических запросов.

Сервер инициализирует класс Context, где хранит необходимое для выполнения запроса окружение: список доступных баз данных, пользователей и прав доступа, настройки, кластеры, список процессов, журнал запросов и т.д. Это окружение используется интерпретаторами.

Мы поддерживаем полную обратную и прямую совместимость для TCP интерфейса: старые клиенты могут общаться с новыми серверами, а новые клиенты могут общаться со старыми серверами. Но мы не хотим поддерживать его вечно и прекращаем поддержку старых версий примерно через год.

***Примечание:*** *Для всех сторонних приложений мы рекомендуем использовать HTTP интерфейс, потому что он прост и удобен в использовании. TCP интерфейс тесно связан с внутренними структурами данных: он использует внутренний формат для передачи блоков данных и использует специальное кадрирование для сжатых данных. Мы не выпустили библиотеку C для этого протокола, потому что потребовала бы линковки большей части кодовой базы RT.WideStore, что непрактично.*

### Выполнение распределенных запросов (Distributed Query Execution)

Сервера в кластере в основном независимы. Вы можете создать Распределенную (Distributed) таблицу на одном или всех серверах в кластере. Такая таблица сама по себе не хранит данные – она только предоставляет возможность "просмотра" всех локальных таблиц на нескольких узлах кластера. При выполнении SELECT распределенная таблица переписывает запрос, выбирает удаленные узлы в соответствии с настройками балансировки нагрузки и отправляет им запрос. Распределенная таблица просит удаленные сервера обработать запрос до той стадии, когда промежуточные результаты с разных серверов могут быть объединены. Затем он получает промежуточные результаты и объединяет их. Распределенная таблица пытается возложить как можно больше работы на удаленные серверы и сократить объем промежуточных данных, передаваемых по сети.

Ситуация усложняется, при использовании подзапросы в случае IN или JOIN, когда каждый из них использует таблицу Distributed. Есть разные стратегии для выполнения таких запросов.

Глобального плана выполнения распределенных запросов не существует. Каждый узел имеет собственный локальный план для своей части работы. У нас есть простое однонаправленное выполнение распределенных запросов: мы отправляем запросы на удаленные узлы и затем объединяем результаты. Но это невозможно для сложных запросов GROUP BY высокой кардинальности или запросов с большим числом временных данных в JOIN: в таких случаях нам необходимо перераспределить («reshuffle») данные между серверами, что требует дополнительной координации. RT.WideStore не поддерживает выполнение запросов такого рода, и нам нужно работать над этим.

### Merge Tree

MergeTree – это семейство движков хранения, поддерживающих индексацию по первичному ключу. Первичный ключ может быть произвольным набором (кортежем) столбцов или выражений. Данные в таблице MergeTree хранятся "частями" (“parts”). Каждая часть хранит данные отсортированные по первичному ключу (данные упорядочены лексикографически). Все столбцы таблицы хранятся в отдельных файлах column.bin в этих частях. Файлы состоят из сжатых блоков. Каждый блок обычно содержит от 64 КБ до 1 МБ несжатых данных, в зависимости от среднего значения размера данных. Блоки состоят из значений столбцов, расположенных последовательно один за другим. Значения столбцов находятся в одинаковом порядке для каждого столбца (порядок определяется первичным ключом), поэтому, когда вы выполняете итерацию по многим столбцам, вы получаете значения для соответствующих строк.

Сам первичный ключ является "разреженным" ("sparse"). Он не относится к каждой отдельной строке, а только к некоторым диапазонам данных. Отдельный файл «primary.idx» имеет значение первичного ключа для каждой N-й строки, где N называется гранулярностью индекса ("index\_granularity", обычно N = 8192). Также для каждого столбца у нас есть файлы column.mrk с "метками" ("marks"), которые обозначают смещение для каждой N-й строки в файле данных. Каждая метка представляет собой пару: смещение начала сжатого блока от начала файла и смещение к началу данных в распакованном блоке. Обычно сжатые блоки выравниваются по меткам, а смещение в распакованном блоке равно нулю. Данные для primary.idx всегда находятся в памяти, а данные для файлов column.mrk кэшируются.

Когда мы собираемся читать что-то из части данных MergeTree, мы смотрим содержимое primary.idx и определяем диапазоны, которые могут содержать запрошенные данные, затем просматриваем содержимое column.mrk и вычисляем смещение, чтобы начать чтение этих диапазонов. Из-за разреженности могут быть прочитаны лишние данные. RT.WideStore не подходит для простых точечных запросов высокой интенсивности, потому что весь диапазон строк размером index\_granularity должен быть прочитан для каждого ключа, а сжатый блок должен быть полностью распакован для каждого столбца. Мы сделали индекс разреженным, потому что мы должны иметь возможность поддерживать триллионы строк на один сервер без существенных расходов памяти на индексацию. Кроме того, поскольку первичный ключ является разреженным, он не уникален: он не может проверить наличие ключа в таблице во время INSERT. Вы можете иметь множество строк с одним и тем же ключом в таблице.

При выполнении INSERT для группы данных в MergeTree, элементы группы сортируются по первичному ключу и образует новую “часть”. Фоновые потоки периодически выбирают некоторые части и объединяют их в одну отсортированную часть, чтобы сохранить относительно небольшое количество частей. Вот почему он называется MergeTree. Конечно, объединение приводит к повышению интенсивности записи. Все части иммутабельные: они только создаются и удаляются, но не изменяются. Когда выполняется SELECT, он содержит снимок таблицы (набор частей). После объединения старые части также сохраняются в течение некоторого времени, чтобы упростить восстановление после сбоя, поэтому, если мы видим, что какая-то объединенная часть, вероятно, повреждена, мы можем заменить ее исходными частями.

MergeTree не является деревом LSM (Log-structured merge-tree — журнально-структурированное дерево со слиянием), потому что оно не содержит «memtable» и «log»: вставленные данные записываются непосредственно в файловую систему. Это делает его пригодным только для вставки данных в пакетах, а не по отдельным строкам и не очень часто - примерно раз в секунду это нормально, а тысячу раз в секунду - нет. Мы сделали это для простоты и потому, что мы уже вставляем данные в пакеты в наших приложениях.

Таблицы MergeTree могут иметь только один (первичный) индекс: вторичных индексов нет. Было бы неплохо разрешить несколько физических представлениям в одной логической таблице, например, хранить данные в более чем одном физическом порядке или даже разрешить представления с предварительно агрегированными данными вместе с исходными данными.

Существуют движки MergeTree, которые выполняют дополнительную работу во время фоновых слияний. Примерами являются CollapsingMergeTree и AggregatingMergeTree. Это можно рассматривать как специальную поддержку обновления. Помните, что это не настоящие обновления, поскольку пользователи обычно не контролируют время выполнения фоновых слияний, а данные в таблице MergeTree почти всегда хранятся в нескольких частях, а не в полностью объединенной форме.

### Репликация

Репликация в RT.WideStore может быть настроена для каждой таблицы отдельно. Вы можете иметь несколько реплицированных и несколько не реплицированных таблиц на одном сервере. Вы также можете реплицировать таблицы по-разному, например, одну с двухфакторной репликацией и другую с трехфакторной.

Репликация реализована в движке таблицы ReplicatedMergeTree. Путь в ZooKeeper указывается в качестве параметра движка. Все таблицы с одинаковым путем в ZooKeeper становятся репликами друг друга: они синхронизируют свои данные и поддерживают согласованность. Реплики можно добавлять и удалять динамически, просто создавая или удаляя таблицу.

Репликация использует асинхронную multi-master схему. Вы можете вставить данные в любую реплику, которая имеет открытую сессию в ZooKeeper, и данные реплицируются на все другие реплики асинхронно. Поскольку RT.WideStore не поддерживает UPDATE, репликация исключает конфликты (conflict-free replication). Поскольку подтверждение вставок кворумом не реализовано, только что вставленные данные могут быть потеряны в случае сбоя одного узла.

Метаданные для репликации хранятся в ZooKeeper. Существует журнал репликации, в котором перечислены действия, которые необходимо выполнить. Среди этих действий: получить часть (get the part); объединить части (merge parts); удалить партицию (drop a partition) и так далее. Каждая реплика копирует журнал репликации в свою очередь, а затем выполняет действия из очереди. Например, при вставке в журнале создается действие «получить часть» (get the part), и каждая реплика загружает эту часть. Слияния координируются между репликами, чтобы получить идентичные до байта результаты. Все части объединяются одинаково на всех репликах. Одна из реплик-лидеров инициирует новое слияние кусков первой и записывает действия «слияния частей» в журнал. Несколько реплик (или все) могут быть лидерами одновременно. Реплике можно запретить быть лидером с помощью merge\_tree настройки replicated\_can\_become\_leader.

Репликация является физической: между узлами передаются только сжатые части, а не запросы. Слияния обрабатываются на каждой реплике независимо, в большинстве случаев, чтобы снизить затраты на сеть, во избежание усиления роли сети. Крупные объединенные части отправляются по сети только в случае значительной задержки репликации.

Кроме того, каждая реплика сохраняет свое состояние в ZooKeeper в виде набора частей и его контрольных сумм. Когда состояние в локальной файловой системе расходится с эталонным состоянием в ZooKeeper, реплика восстанавливает свою согласованность путем загрузки отсутствующих и поврежденных частей из других реплик. Когда в локальной файловой системе есть неожиданные или испорченные данные, RT.WideStore не удаляет их, а перемещает в отдельный каталог и забывает об этом.

***Примечание:*** *Кластер RT.WideStore состоит из независимых шардов, а каждый шард состоит из реплик. Кластер* ***не является эластичным*** *(not elastic), поэтому после добавления нового шарда данные не будут автоматически распределены между ними. Вместо этого нужно изменить настройки, чтобы выровнять нагрузку на кластер. Эта реализация дает вам больший контроль, и вполне приемлема для относительно небольших кластеров, таких как десятки узлов. Но для кластеров с сотнями узлов, которые мы используем в эксплуатации, такой подход становится существенным недостатком. Движки таблиц, которые охватывают весь кластер с динамически реплицируемыми областями, которые могут быть автоматически разделены и сбалансированы между кластерами, еще предстоит реализовать.*

## Интерфейсы взаимодействия с Системой

RT.WideStore предоставляет два сетевых интерфейса (оба могут быть дополнительно обернуты в TLS для дополнительной безопасности):

* [HTTP](https://clickhouse.tech/docs/ru/interfaces/http/), который задокументирован и прост для использования напрямую.
* [Native TCP](https://clickhouse.tech/docs/ru/interfaces/tcp/), который имеет меньше накладных расходов.

В большинстве случаев рекомендуется использовать подходящий инструмент или библиотеку, а не напрямую взаимодействовать с RT.WideStore по сути.

Поддерживаются:

* [консольный клиент](https://clickhouse.tech/docs/ru/interfaces/cli/),
* [JDBC-драйвер](https://clickhouse.tech/docs/ru/interfaces/jdbc/),
* [ODBC-драйвер](https://clickhouse.tech/docs/ru/interfaces/odbc/),
* [C++ клиентская библиотека](https://clickhouse.tech/docs/ru/interfaces/cpp/).

Существует также широкий спектр сторонних библиотек для работы с RT.WideStore:

* [клиентские библиотеки](https://clickhouse.tech/docs/ru/interfaces/third-party/client-libraries/),
* [библиотеки для интеграции](https://clickhouse.tech/docs/ru/interfaces/third-party/integrations/),
* [визуальные интерфейсы](https://clickhouse.tech/docs/ru/interfaces/third-party/gui/).

### HTTP-интерфейс

HTTP интерфейс позволяет использовать RT.WideStore на любой платформе, из любого языка программирования. HTTP интерфейс удобно использовать для работы из Java и Perl, а также из shell-скриптов. HTTP интерфейс можно использовать из Perl, Python и Go. HTTP интерфейс более ограничен по сравнению с родным интерфейсом, но является более совместимым.

Для уменьшения трафика по сети можно использовать сжатие при передаче большого количества данных, а также для создания сразу сжатых дампов.

Вы можете использовать внутренний формат сжатия RT.WideStore при передаче данных. Формат сжатых данных нестандартный, и вам придётся использовать для работы с ним специальную программу clickhouse-compressor.

Также можно использовать [сжатие HTTP](https://en.wikipedia.org/wiki/HTTP_compression). RT.WideStore поддерживает следующие [методы сжатия](https://en.wikipedia.org/wiki/HTTP_compression#Content-Encoding_tokens):

* gzip,
* br,
* deflate,
* xz.

### [Native TCP](https://clickhouse.tech/docs/ru/interfaces/tcp/)

Нативный протокол используется в [клиенте командной строки](https://clickhouse.tech/docs/ru/interfaces/cli/), для взаимодействия между серверами во время обработки распределенных запросов, а также в других программах на C++. К сожалению, у родного протокола RT.WideStore пока нет формальной спецификации, но в нем можно разобраться с использованием исходного кода RT.WideStore (начиная с [примерно этого места](https://github.com/ClickHouse/ClickHouse/tree/master/src/Client)) и/или путем перехвата и анализа TCP трафика.

### [Консольный клиент](https://clickhouse.tech/docs/ru/interfaces/cli/)

RT.WideStore предоставляет собственный клиент командной строки: clickhouse-client. Клиент поддерживает запуск с аргументами командной строки и с конфигурационными файлами.

Клиенты и серверы различных версий совместимы, однако если клиент старее сервера, то некоторые новые функции могут быть недоступны. Мы рекомендуем использовать одинаковые версии клиента и сервера.

Клиент может быть использован в интерактивном и не интерактивном (batch) режиме.

### [JDBC-драйвер](https://clickhouse.tech/docs/ru/interfaces/jdbc/)

Существует следующие [JDBC-драйвер](https://clickhouse.tech/docs/ru/interfaces/jdbc/)ы:

* Официальный драйвер – это базовая и ограниченная реализация драйвера jdbc для RT.WideStore. Он поддерживает минимальный набор функций, которые можно использовать;
* Драйверы от сторонних организаций:
* ClickHouse-Native-JDBC – собственная библиотека JDBC для доступа к RT.WideStore в Java также предоставляет библиотеку для интеграции с Apache Spark;
* clickhouse4j – более легкая и быстрая альтернатива официальному JDBC-драйверу RT.WideStore.

### [ODBC-драйвер](https://clickhouse.tech/docs/ru/interfaces/odbc/)

Это официальная реализация драйвера ODBC для доступа к RT.WideStore как источнику данных.

### [C++ клиентская библиотека](https://clickhouse.tech/docs/ru/interfaces/cpp/)

Библиотека к C++ которая поддерживает следующие типы данных:

* массив (T),
* дата,
* DateTime, DateTime64,
* DateTime ([часовой пояс]), DateTime64 (N, [часовой пояс]),
* десятичное32, десятичное64, десятичное128;
* Enum8, Enum16,
* фиксированная строка (N),
* Float32, Float64,
* IPv4, IPv6,
* обнуляемый (T),
* нить,
* кортеж,
* UInt8, UInt16, UInt32, UInt64, Int8, Int16, Int32, Int64.

### [Клиентские библиотеки](https://clickhouse.tech/docs/ru/interfaces/third-party/client-libraries/)

Существуют следующие клиентские библиотеки для доступа к RT.WideStore как источнику данных:

* Python:
* [infi.clickhouse\_orm](https://github.com/Infinidat/infi.clickhouse_orm),
* [clickhouse-driver](https://github.com/mymarilyn/clickhouse-driver),
* [clickhouse-client](https://github.com/yurial/clickhouse-client),
* [aiochclient](https://github.com/maximdanilchenko/aiochclient),
* [asynch](https://github.com/long2ice/asynch).
* PHP:
* [smi2/phpclickhouse](https://packagist.org/packages/smi2/phpClickHouse),
* [8bitov/clickhouse-php-client](https://packagist.org/packages/8bitov/clickhouse-php-client),
* [bozerkins/clickhouse-client](https://packagist.org/packages/bozerkins/clickhouse-client),
* [simpod/clickhouse-client](https://packagist.org/packages/simpod/clickhouse-client),
* [seva-code/php-click-house-client](https://packagist.org/packages/seva-code/php-click-house-client),
* [SeasClick C++ client](https://github.com/SeasX/SeasClick),
* [glushkovds/phpclickhouse-laravel](https://packagist.org/packages/glushkovds/phpclickhouse-laravel),
* [kolya7k ClickHouse PHP extension](https://github.com/kolya7k/clickhouse-php).
* Go:
* [clickhouse](https://github.com/kshvakov/clickhouse/),
* [go-clickhouse](https://github.com/roistat/go-clickhouse),
* [mailrugo-clickhouse](https://github.com/mailru/go-clickhouse),
* [golang-clickhouse](https://github.com/leprosus/golang-clickhouse).
* NodeJs:
* [clickhouse (NodeJs)](https://github.com/TimonKK/clickhouse),
* [node-clickhouse](https://github.com/apla/node-clickhouse).
* Perl:
* [perl-DBD-ClickHouse](https://github.com/elcamlost/perl-DBD-ClickHouse),
* [HTTP-ClickHouse](https://metacpan.org/release/HTTP-ClickHouse),
* [AnyEvent-ClickHouse](https://metacpan.org/release/AnyEvent-ClickHouse).
* Ruby:
* [ClickHouse (Ruby)](https://github.com/shlima/click_house),
* [clickhouse-activerecord](https://github.com/PNixx/clickhouse-activerecord).
* R:
* [clickhouse-r](https://github.com/hannesmuehleisen/clickhouse-r),
* [RClickhouse](https://github.com/IMSMWU/RClickhouse),
* Java:
* [clickhouse-client-java](https://github.com/VirtusAI/clickhouse-client-java).
* Scala:
* [clickhouse-scala-client](https://github.com/crobox/clickhouse-scala-client).
* Kotlin:
* [AORM](https://github.com/TanVD/AORM).
* C#:
* [Octonica.ClickHouseClient](https://github.com/Octonica/ClickHouseClient),
* [ClickHouse.Ado](https://github.com/killwort/ClickHouse-Net),
* [ClickHouse.Client](https://github.com/DarkWanderer/ClickHouse.Client),
* [ClickHouse.Net](https://github.com/ilyabreev/ClickHouse.Net).
* Elixir:
* [clickhousex](https://github.com/appodeal/clickhousex/),
* [pillar](https://github.com/sofakingworld/pillar).
* Nim:
* [nim-clickhouse](https://github.com/leonardoce/nim-clickhouse).

### [Библиотеки для интеграции](https://clickhouse.tech/docs/ru/interfaces/third-party/integrations/)

Существуют следующие библиотеки для интеграции к RT.WideStore от сторонних разработчиков:

1. Инфраструктурные продукты
   * Реляционные системы управления базами данных:
   * [MySQL](https://www.mysql.com/):
     + [mysql2ch](https://github.com/long2ice/mysql2ch),
     + [ProxySQL](https://github.com/sysown/proxysql/wiki/ClickHouse-Support),
     + [clickhouse-mysql-data-reader](https://github.com/Altinity/clickhouse-mysql-data-reader),
     + [horgh-replicator](https://github.com/larsnovikov/horgh-replicator).
   * [PostgreSQL](https://www.postgresql.org/):
     + [clickhousedb\_fdw](https://github.com/Percona-Lab/clickhousedb_fdw),
     + [infi.clickhouse\_fdw](https://github.com/Infinidat/infi.clickhouse_fdw) (использует [infi.clickhouse\_orm](https://github.com/Infinidat/infi.clickhouse_orm)),
     + [pg2ch](https://github.com/mkabilov/pg2ch),
     + [clickhouse\_fdw](https://github.com/adjust/clickhouse_fdw).
   * [MSSQL](https://en.wikipedia.org/wiki/Microsoft_SQL_Server):
     + [ClickHouseMightrator](https://github.com/zlzforever/ClickHouseMigrator).
   * Очереди сообщений:
   * [Kafka](https://kafka.apache.org/):
     + [clickhouse\_sinker](https://github.com/housepower/clickhouse_sinker) (использует [Go client](https://github.com/ClickHouse/clickhouse-go/)),
     + [stream-loader-clickhouse](https://github.com/adform/stream-loader).
   * Потоковая обработка:
   * [Flink](https://flink.apache.org/):
     + [flink-clickhouse-sink](https://github.com/ivi-ru/flink-clickhouse-sink).
   * Хранилища объектов:
   * [S3](https://en.wikipedia.org/wiki/Amazon_S3):
     + [clickhouse-backup](https://github.com/AlexAkulov/clickhouse-backup).
   * Оркестрация контейнеров:
   * [Kubernetes](https://kubernetes.io/):
     + [clickhouse-operator](https://github.com/Altinity/clickhouse-operator).
   * Системы управления конфигурацией:
   * [puppet](https://puppet.com/):
     + [innogames/clickhouse](https://forge.puppet.com/innogames/clickhouse),
     + [mfedotov/clickhouse](https://forge.puppet.com/mfedotov/clickhouse).
   * Мониторинг:
   * [Graphite](https://graphiteapp.org/):
     + [graphouse](https://github.com/yandex/graphouse),
     + [carbon-clickhouse](https://github.com/lomik/carbon-clickhouse) +,
     + [graphite-clickhouse](https://github.com/lomik/graphite-clickhouse),
     + [graphite-ch-optimizer](https://github.com/innogames/graphite-ch-optimizer) – оптимизирует партиции таблиц [\*GraphiteMergeTree](https://clickhouse.tech/docs/ru/single/#graphitemergetree) согласно правилам в [конфигурации rollup](https://clickhouse.tech/docs/ru/single/#rollup-configuration).
   * [Grafana](https://grafana.com/):
     + [clickhouse-grafana](https://github.com/Vertamedia/clickhouse-grafana).
   * [Prometheus](https://prometheus.io/):
     + [clickhouse\_exporter](https://github.com/f1yegor/clickhouse_exporter),
     + [PromHouse](https://github.com/Percona-Lab/PromHouse),
     + [clickhouse\_exporter](https://github.com/hot-wifi/clickhouse_exporter) (использует [Go client](https://github.com/kshvakov/clickhouse/)).
   * [Nagios](https://www.nagios.org/):
     + [check\_clickhouse](https://github.com/exogroup/check_clickhouse/),
     + [check\_clickhouse.py](https://github.com/innogames/igmonplugins/blob/master/src/check_clickhouse.py).
   * [Zabbix](https://www.zabbix.com/):
     + [clickhouse-zabbix-template](https://github.com/Altinity/clickhouse-zabbix-template).
   * [Sematext](https://sematext.com/):
     + [clickhouse интеграция](https://github.com/sematext/sematext-agent-integrations/tree/master/clickhouse).
   * Логирование:
   * [rsyslog](https://www.rsyslog.com/):
     + [omclickhouse](https://www.rsyslog.com/doc/master/configuration/modules/omclickhouse.html).
   * [fluentd](https://www.fluentd.org/):
     + [loghouse](https://github.com/flant/loghouse) (для [Kubernetes](https://kubernetes.io/)).
   * [Sematext](https://www.sematext.com/logagent):
     + [logagent output-plugin-clickhouse](https://sematext.com/docs/logagent/output-plugin-clickhouse/).
   * Гео:
   * [MaxMind](https://dev.maxmind.com/geoip/):
     + [clickhouse-maxmind-geoip](https://github.com/AlexeyKupershtokh/clickhouse-maxmind-geoip).
2. Экосистемы вокруг языков программирования
   * Python:
   * [SQLAlchemy](https://www.sqlalchemy.org/):
     + [sqlalchemy-clickhouse](https://github.com/cloudflare/sqlalchemy-clickhouse) (использует [infi.clickhouse\_orm](https://github.com/Infinidat/infi.clickhouse_orm)).
   * [pandas](https://pandas.pydata.org/):
     + [pandahouse](https://github.com/kszucs/pandahouse).
   * PHP:
   * [Doctrine](https://www.doctrine-project.org/):
     + [dbal-clickhouse](https://packagist.org/packages/friendsofdoctrine/dbal-clickhouse).
   * R:
   * [dplyr](https://db.rstudio.com/dplyr/),
   * [RClickhouse](https://github.com/IMSMWU/RClickhouse) (использует [clickhouse-cpp](https://github.com/artpaul/clickhouse-cpp)).
   * Java:
   * [Hadoop](http://hadoop.apache.org/),
   * [clickhouse-hdfs-loader](https://github.com/jaykelin/clickhouse-hdfs-loader) (использует [JDBC](https://clickhouse.tech/docs/ru/single/#../../sql-reference/table-functions/jdbc/)).
   * Scala:
   * [Akka](https://akka.io/),
   * [clickhouse-scala-client](https://github.com/crobox/clickhouse-scala-client).
   * C#
   * [ADO.NET](https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ado-net-overview),
   * [ClickHouse.Ado](https://github.com/killwort/ClickHouse-Net),
   * [ClickHouse.Client](https://github.com/DarkWanderer/ClickHouse.Client),
   * [ClickHouse.Net](https://github.com/ilyabreev/ClickHouse.Net),
   * [ClickHouse.Net.Migrations](https://github.com/ilyabreev/ClickHouse.Net.Migrations).
   * Elixir:
   * [Ecto](https://github.com/elixir-ecto/ecto),
   * [clickhouse\_ecto](https://github.com/appodeal/clickhouse_ecto).
   * Ruby:
   * [Ruby on Rails](https://rubyonrails.org/):
     + [activecube](https://github.com/bitquery/activecube),
     + [ActiveRecord](https://github.com/PNixx/clickhouse-activerecord).
   * [GraphQL](https://github.com/graphql):
     + [activecube-graphql](https://github.com/bitquery/activecube-graphql).

### [Визуальные интерфейсы](https://clickhouse.tech/docs/ru/interfaces/third-party/gui/)

Визуальные интерфейсы от сторонних разработчиков:

1. ***С открытым исходным кодом***

**Tabix**

Веб-интерфейс для RT.WideStore в проекте [Tabix](https://github.com/tabixio/tabix).

Основные возможности:

* Работает с RT.WideStore напрямую из браузера, без необходимости установки дополнительного ПО.
* Редактор запросов с подсветкой синтаксиса.
* Автодополнение команд.
* Инструменты графического анализа выполнения запросов.
* Цветовые схемы на выбор.

**HouseOps**

UI/IDE для OSX, Linux и Windows.

Основные возможности:

* Построение запросов с подсветкой синтаксиса.
* Просмотр ответа в табличном или JSON представлении.
* Экспортирование результатов запроса в формате CSV или JSON.
* Список процессов с описанием.
* Режим записи.
* Возможность остановки (KILL) запроса.
* Граф базы данных. Показывает все таблицы и их столбцы с дополнительной информацией.
* Быстрый просмотр размера столбца.
* Конфигурирование сервера.

Планируется разработка следующих возможностей:

* Управление базами.
* Управление пользователями.
* Анализ данных в режиме реального времени.
* Мониторинг кластера.
* Управление кластером.
* Мониторинг реплицированных и Kafka таблиц.

**LightHouse**

Легковесный веб-интерфейс для RT.WideStore.

Основные возможности:

* Список таблиц с фильтрацией и метаданными.
* Предварительный просмотр таблицы с фильтрацией и сортировкой.
* Выполнение запросов только для чтения.

**Redash**

Платформа для отображения данных.

Поддерживает множество источников данных, включая RT.WideStore. Redash может объединять результаты запросов из разных источников в финальный набор данных.

Основные возможности:

* Мощный редактор запросов.
* Проводник по базе данных.
* Инструменты визуализации, позволяющие представить данные в различных формах.

**DBeaver**

Универсальный desktop клиент баз данных с поддержкой RT.WideStore.

Основные возможности:

* Построение запросов с подсветкой синтаксиса.
* Просмотр таблиц.
* Автодополнение команд.
* Полнотекстовый поиск.

**clickhouse-cli**

Альтернативный клиент командной строки для RT.WideStore, написанный на Python 3.

Основные возможности:

* Автодополнение.
* Подсветка синтаксиса для запросов и вывода данных.
* Поддержка постраничного просмотра для результирующих данных.
* Дополнительные PostgreSQL-подобные команды.

**clickhouse-flamegraph**

Специализированный инструмент для визуализации system.trace\_log в виде [flamegraph](http://www.brendangregg.com/flamegraphs.html).

**clickhouse-plantuml**

Скрипт, генерирующий [PlantUML](https://plantuml.com/) диаграммы схем таблиц.

**xeus-clickhouse**

Ядро Jupyter для RT.WideStore, которое поддерживает запрос RT.WideStore-данных с использованием SQL в Jupyter.

1. ***Коммерческие:***

**DataGrip**

IDE для баз данных о JetBrains с выделенной поддержкой RT.WideStore. Он также встроен в другие инструменты на основе IntelliJ: PyCharm, IntelliJ IDEA, GoLand, PhpStorm и другие.

Основные возможности:

* Очень быстрое дополнение кода.
* Подсветка синтаксиса для SQL диалекта RT.WideStore.
* Поддержка функций, специфичных для RT.WideStore, например, вложенных столбцов, движков таблиц.
* Редактор данных.
* Рефакторинги.
* Поиск и навигация.

**Holistics Software**

full-stack платформа для обработки данных и бизнес-аналитики.

Основные возможности:

* Автоматизированные отчёты на почту, Slack, и Google Sheet.
* Редактор SQL c визуализацией, контролем версий, автодополнением, повторным использованием частей запроса и динамическими фильтрами.
* Встроенные инструменты анализа отчётов и всплывающие (iframe) дашборды.
* Подготовка данных и возможности ETL.
* Моделирование данных с помощью SQL для их реляционного отображения.

**Looker**

Платформа для обработки данных и бизнес-аналитики. Поддерживает более 50 диалектов баз данных, включая RT.WideStore. Looker можно установить самостоятельно или воспользоваться готовой платформой SaaS.

Просмотр данных, построение отображений и дашбордов, планирование отчётов и обмен данными с коллегами доступны с помощью браузера. Также, Looker предоставляет ряд инструментов, позволяющих встраивать сервис в другие приложения и API для обмена данными.

Основные возможности:

* Язык LookML, поддерживающий [моделирование данных](https://looker.com/platform/data-modeling).
* Интеграция с различными системами с помощью [Data Actions](https://looker.com/platform/actions).
* Инструменты для встраивания сервиса в приложения.
* API.

# УСТАНОВКА RT.WIDESTORE

## Системные требования

RT.WideStore может работать на любой операционной системе Linux, FreeBSD или Mac OS X с архитектурой процессора x86\_64, AArch64 или PowerPC64LE.

Предварительно собранные пакеты компилируются для x86\_64 и используют набор инструкций SSE 4.2, поэтому, если не указано иное, его поддержка в используемом процессоре, становится дополнительным требованием к системе. Вот команда, чтобы проверить, поддерживает ли текущий процессор SSE 4.2:

grep -q sse4\_2 /proc/cpuinfo && echo "SSE 4.2 supported" || echo "SSE 4.2 not supported"

Чтобы запустить RT.WideStore на процессорах, которые не поддерживают SSE 4.2, либо имеют архитектуру AArch64 или PowerPC64LE, необходимо самостоятельно [собрать RT.WideStore из исходного кода](https://clickhouse.tech/docs/ru/getting-started/install/#from-sources) с соответствующими настройками конфигурации (см.п. 5.2.6).

## Доступные варианты установки

### Из Tgz архивов

Для всех дистрибутивов, где невозможна установка deb и rpm пакетов рекомендуется использовать предкомпилированные бинарники из tgz архивов.

Интересующую версию архивов можно скачать вручную с помощью curl или wget из репозитория http://widestore.rt.ru/clickhouse-v21.3.13.9.tar.gz.

После этого архивы нужно распаковать и запустить сервер:

./clickhouse-server start

### Из Docker образа

Для запуска RT.WideStore в Docker нужно следовать инструкции на [Docker Hub](https://hub.docker.com/r/yandex/clickhouse-server/). Внутри образов используются официальные deb пакеты.

# ЗАПУСК RT.WIDESTORE

Для запуска сервера в качестве демона, выполните:

sudo service clickhouse-server start

Смотрите логи в директории:

/var/log/clickhouse-server/.

Если сервер не стартует, проверьте корректность конфигурации в файле:

/etc/clickhouse-server/config.xml

Также можно запустить сервер вручную из консоли:

clickhouse-server --config-file=/etc/clickhouse-server/config.xml

При этом, лог будет выводиться в консоль, что удобно для разработки.

Если конфигурационный файл лежит в текущей директории, то указывать параметр "--config-file" не требуется, по умолчанию будет использован файл "./config.xml".

После запуска сервера, соединиться с ним можно с помощью клиента командной строки:

clickhouse-client

По умолчанию он соединяется с localhost:9000, от имени пользователя default без пароля. Также клиент может быть использован для соединения с удалённым сервером с помощью аргумента "--host".

Терминал должен использовать кодировку UTF-8.

Более подробная информация о клиенте располагается в разделе [«Клиент командной строки»](https://clickhouse.tech/docs/ru/interfaces/cli/).

Пример проверки работоспособности системы:

$ ./clickhouse-client

ClickHouse client version 0.0.18749.

Connecting to localhost:9000.

Connected to ClickHouse server version 0.0.18749.

:) SELECT 1

SELECT 1

┌─1─┐

│ 1 │

└───┘

1 rows in set. Elapsed: 0.003 sec.