PARTITION BY, ORDER BY, TTL, типы колонок, кодеки, вторичные индексы

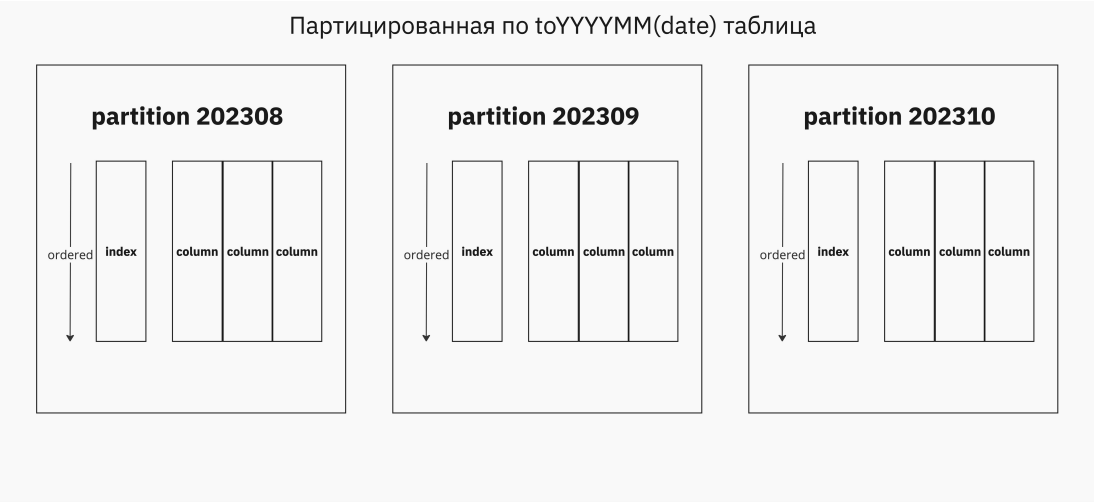
Как получить размер таблицы, столбцов, партиций

Параметры *MergeTree таблиц

PARTITION BY

Модификатор **`PARTITION BY`** физически разделяет файлы таблицы на части – партиции. Это помогает избегать полного сканирования таблицы.

Например, если в таблице партицирование по месяцу, то при запросах будут читаться только партиции, которые попадают под фильтр по дате.



> **Размер партиции должен быть от 5 GB до 15 GB**
>
> Если таблица весит < 5 GB – партиционировать не стоит (**`PARTITION BY`** `tuple()`)
>
> Чем больше партиция, тем больше шанс, что она хорошо сожмется при выборе правильной сортировки.
>
> Для таблицы с данными за 2 года размером 10 GB партиционирование по дню – плохой вариант, получится очень много маленьких партиций. Будут дополнительные затраты на открытие каждого файла, и есть шанс, что они плохо сожмутся.
> Такое же правило для 1 GB таблицы с данными за 2 года с партиционированием по месяцу.

ORDER BY

Ключ сортировки **`ORDER BY`**:

- Отвечает за то, как данные (строки таблиц) будут отсортированы внутри каждой партиции.
- Является индексом таблицы.
  (Принципиально отличается от индекса в PostgreSQL. В ClickHouse индекс разреженный, т.е. несколько строк могут иметь одинаковый индекс.
  Указывает на часть данных, где может находиться нужная строка)

  Помогает ClickHouse не фуллсканить таблицу, для нахождения строки, а читать только нужные блоки данных.

Выбор правильного **`ORDER BY`** позволяет сильно снизить занимаемое место на диске и ускорить запросы.
ClickHouse очень хорошо сжимает данные, если похожие строки идут друг за другом.
Подробно про ключи сортировки можно прочитать в документации.

Как выбрать ORDER BY:

- **Занимать меньше места на диске, и, возможно, ускорить SELECT.**
  Нужно подобрать ключ сортировки таким образом, чтобы похожие строки шли по порядку. Если похожие значения в колонках физически находятся рядом, то ClickHouse эффективно сожмет такие столбцы. Особенно эффект будет заметен, если одинаковые значения тяжелых колонок находятся рядом.
- **Можно дополнительно: Ускорить WHERE**

  Поставить на первое место ключа сортировки столбец, который часто участвует в **`WHERE`**.
  Однако, стоит сравнить, на сколько сильно увеличится размер таблицы, по сравнению с сортировкой, которая максимально сжимает данные таблицы.

В первую очередь обратить внимание на:

- Природу данных. По какому полю отсортировать, чтобы в таблице похожие строки находились физически рядом; насколько в колонке много уникальных значений.
- Размер колонок на семпле данных.

Примеры выбора ORDER BY из практики (обязательно к ознакомлению).

*Подробная статья из документации про то как работают индексы.

TTL

Это параметр, который определяет время жизни строк в таблице.
TTL вешается на поле с типом Date/DateTime. ClickHouse удалит устаревшие строки сам в фоне.
Подробнее в документации.

Желательно ставить TTL на таблицы, это избавит от процесса очистки старых строк и сэкономит место на диске.

DateTime столбец в аргументе к TTL обязательно нужно приводить к Date.
Иначе ClickHouse начнет запускать процесс удаления старых строк очень часто, что приведет к большой нагрузке.
Были кейсы, когда из-за этого производительность всего кластера деградировала.

Пример

```
TTL toDate(created_at) + toIntervalMonth(3)
```

```
CREATE TABLE sandbox.universal_locations ON CLUSTER shard_group_old
(
    `shopper_id` Int32,
    `city` String,
    `role` String,
    `context_device_manufacturer` String,
    `context_app_version` String,
    `context_os_version` String,
    `user_uuid` UUID,
    `transport` String,
    `location_id` String,
    `distance_lag` Nullable(Float32),
    `created_at` DateTime(),
    `time_lag` Nullable(Float32),
    `shift_id` UInt32,
    `fact_started_at` Nullable(DateTime('UTC')),
    `fact_ended_at` Nullable(DateTime('UTC')),
    `during_shipment_flg` Nullable(UInt8),
    `is_fake_gps_app_detected` Nullable(UInt8)
)
ENGINE = ReplicatedMergeTree('/tables/{shard}/sandbox/universal_locations', '{replica}')
PARTITION BY toYYYYMMDD(created_at)
ORDER BY created_at
TTL toDate(created_at) + toIntervalMonth(3) /* <--------- */
SETTINGS index_granularity = 8192
```

Строки с toDate(created_at) старше трех месяцев будут автоматически удаляться.

```
ALTER TABLE db.table_name MODIFY TTL ttl_expression;
```

```
ALTER TABLE sandbox.universal_locations MODIFY TTL toDate(created_at) + toIntervalMonth(3);
```

Применение настройки ttl_only_drop_parts

Настройку следует применять на таблицу, если выполняются следующие условия:

1. Всегда, если партиционирование совпадает с периодом TTL. То есть TTL всегда будет удалять целые партиции и не строки внутри одной из партиций. Например,
    a. `partition by toYYYYMM(date_column) & TTL toStartOfMonth(date_column) + toIntervalMonth(2) <-----` тут условие TTL срабатывает раз в месяц, удаляя всю партицию с месяцем данных
    b. `partition by toDate(date_column) & TTL toDate(date_column) + toIntervalMonth(2) <-----   TTL`
2. Таблица партиционирована по месяцу или дню. То есть не `tuple()` и не `YEAR(date_column)`.
3. `1 .`

Типы колонок

Нужно использовать минимально возможные типы данных

- Использовать **UUID**, где возможно. Подробнее про UUID.
  UUID значение с типом **String** весит 37 байт, а с типом **UUID** 16 байт, то есть в 2 раза меньше.
- Если колонку можно сделать числом - делать ее числом.
- Подбирать числовые типы из нужного диапазона.
  Например, использовать **UInt16**, вместо **Int64**, когда это возможно. На хранение одного значения **UInt16** требуется 2 байта, для **Int64** требуется 8 байт. В этом случае столбец будет весить в 4 раза меньше, дополнительно, операции с ним будут работать быстрее.
  Или же вместо **Float64** использовать **Float32**, когда это возможно, например, когда в колонке хранятся доли. **Float32** весит в 2 раза меньше, чем **Float64** – 4 байта против 8 байт на значение.
- Строки с датами приводить к Date или DateTime.

LowCardinality

LowCardinality — это надстройка, изменяющая способ хранения и правила обработки данных.
Под капотом создает словарь возможных значений столбца, а в столбце хранит только номер значения. Внешне, колонка никак не меняется.

Тем самым, LowCardinality может сжать нечисловой столбец с низкой кардинальностью в десятки раз, также агрегации с этим столбцом будут работать быстрее.

Стоит использовать, если в колонке высокая доля повторяющихся значений и количество уникальных значений менее 10000.
Например, поля, которые хранят категории товаров, города и т.д.

Как проверить количество уникальных строк в столбце

```
SELECT
    uniqHLL12(field_name)
FROM schema.table_name
```

Статья от Altinity про LowCardinality.

Enum

Отличие Enum от LowCardinality в том, что словарь значений задается пользователем на уровне DDL. Приоритетнее использовать Enum, чем LowCardinality, если это возможно, например, когда перечень возможных значений определен и останется неизменным.

Nullable

Стоит использовать Nullable только там, где это необходимо.
Поскольку его использование увеличивает размер колонки и снижает производительность. Подробнее в документации.
Для Nullable поля ClickHouse создает отдельный файл с масками Null.

У всех колонок есть дефолтные значения, например для String это пустая строка, для численных типов - 0.
Есть вставить Null в не Nullable колонку, то ClickHouse не поднимет ошибку, а вместо Null вставит дефолтное значение.
Дефолтное значение можно переопределить, например:

```
CREATE TABLE ...
(
    some_column String DEFAULT 'some_value'
)
...
```

Пример того, как выбор оптимальных типов столбцов может сжать размер таблицы

Возьмём часть данных из таблицы "sandbox.universal_locations". В тестовой выборке 1.2 млрд строк.

Сравним размер полей с неоптимальными и оптимальными типоами данных.

Поле	Примерное количество уникальных значений	Неоптимальный тип	Размер поля с неоптимальным типом	Размер поля с оптимальным типом	Оптимальный тип	Степень сжатия, если использовать оптимальный тип
location_id	1 251 198 538	String	43.41 GiB	18.84 GiB	UUID	2.3
distance_lag	2 003 607	Nullable(Float64)	6.51 GiB	4.54 GiB	Nullable(Float32)	1.44
time_lag	1 300 122	Nullable(Float64)	6.23 GiB	4.63 GiB	Nullable(Float32)	1.35
user_uuid	72 401	String	218.98 MiB	99.41 MiB	UUID	2.2
city	308	String	111.95 MiB	13.75 MiB	LowCardinality(String)	8.14
context_device_manufacturer	109	String	44.38 MiB	9.18 MiB	LowCardinality(String)	4.83
context_app_version	27	String	42.22 MiB	8.81 MiB	LowCardinality(String)	4.79

role	3	String	36.06 MiB	5.91 MiB	LowCardinality(String)	6.1
context_os_version	92	String	36.47 MiB	9.17 MiB	LowCardinality(String)	3.98
transport	3	String	34.21 MiB	9.14 MiB	LowCardinality(String)	3.74

Как видно, используя оптимальные типы колонок, можно сжать таблицу. **57 GiB 38 GIB, занимаемое место уменьшилось в 1.5 раз**.
В оптимальной версии таблицы использовались:

- UUID где возможно. Самое объемное поле "location_id", которое весило 43 GiB при использовании типа String, теперь занимает 18.84 GiB при хранении в UUID.
- LowCardinality для колонок с низкой кардинальностью. LowCardinality сократило размер таких колонок в 4-8 раз.
- Float32 вместо Float64. Если Float32 достаточно, то его использование вместо Float64 может сэкономить ~ 2 GiB; как получилось с полями "distance_lag", "time_lag".

Компрессия и кодеки

Из документации ClickHouse

Кодеки:

- `NONE` — без сжатия. (Не использовать)
- `LZ4` — алгоритм сжатия без потерь используемый по умолчанию. Применяет быстрое сжатие LZ4.
- `LZ4HC[(level)]` — алгоритм LZ4 HC (high compression) с настраиваемым уровнем сжатия. Уровень по умолчанию — 9. Настройка `level <= 0` устанавливает уровень сжания по умолчанию. Возможные уровни сжатия: [1, 12]. Рекомендуемый диапазон уровней: [4, 9].
- `ZSTD[(level)]` — алгоритм сжатия ZSTD с настраиваемым уровнем сжатия `level`. Возможные уровни сжатия: [1, 22]. Уровень сжатия по умолчанию: 1.

Высокие уровни сжатия полезны для ассимметричных сценариев, подобных «один раз сжал, много раз распаковал». Они подразумевают лучшее сжатие, но большее использование CPU.

> В нашем ClickHouse кластере ко всем колонкам по умолчанию задается LZ4 компрессия.

При понимании природы данных, для сжатия можно использовать кодеки. *Однако, перед этим стоит выбрать правильные типы колонок, сортировку и партиционирование.*
Применение кодеков имеет смысл, если эффект сжатия больше 30%.

Встречаемые в практике кодеки:

- **Delta**, **DoubleDelta** - для монотонных последовательностей.
- **T64** - для целочисленных типов. Отрезает неиспользованные биты. Эффективно, когда все данные агрегированны вокруг среднего с небольшой дисперсией
  Например min = 10.000, max = 60.000, avg = 30.000, type = Int32

Статья от Altinity с сравнением кодеков

Skip Index

Вторичные индексы предназначены для оптимизации запросов к таблице, когда индекс **`ORDER BY`** используется неэффективно.
Как и **`ORDER BY`** позволяет читать меньше данных с диска, если это возможно.
Важно понимать, что реализация вторичных индексов принципиально отличается от индексов в OLTP базах.

В документации есть подробная статья с описанием, примерами и советами, на что стоит опираться, при использовании skip индексов.

Любое создание вторичных индексов должно быть согласовано с DWH, потому, что неправильно выставленный индекс хуже, чем вообще никакой.
На практике используются редко. Это последний шаг оптимизации запросов к таблице, вначале стоит выбрать правильные типы колонок, сортировку и партиционирование.