Yandex

Yandex

Dive into ClickHouse Storage System

Alexander Sapin, Software Engineer

ClickHouse Storages

External Table Engines:

- File on FS
- S3
- HDFS
- MySQL
- ...

Internal Table Engines:

- Memory
- Log
- StripeLog
- MergeTree family

ClickHouse Storages

External Table Engines:

- File on FS
- S3
- HDFS
- MySQL
- ...

Internal Table Engines:

- Memory
- Log
- StripeLog
- **MergeTree family**

MergeTree Engines Family

Advantages:

- Inserts are atomic
- Selects are blazing fast
- Primary and secondary indexes
- Data modification!
- Inserts and selects don’t block each other

MergeTree Engines Family

Advantages:

- Inserts are atomic
- Selects are blazing fast
- Primary and secondary indexes
- Data modification!
- Inserts and selects don’t block each other

Features:

- Infrequent INSERTs required

MergeTree Engines Family

Advantages:

- Inserts are atomic
- Selects are blazing fast
- Primary and secondary indexes
- Data modification!
- Inserts and selects don’t block each other

Features:

- Infrequent INSERTs required (work in progress)

MergeTree Engines Family

Advantages:

- Inserts are atomic
- Selects are blazing fast
- Primary and secondary indexes
- Data modification!
- Inserts and selects don’t block each other

Features:

- Infrequent INSERTs required (work in progress)
- Background operations on records with the same keys
- Primary key is NOT unique

Write

Create Table and Fill Some Data

Create Table:

```
CREATE TABLE mt (
    EventDate Date,
    OrderID Int32,
    BannerID UInt64,
    GoalNum Int8
) ENGINE = MergeTree()
PARTITION BY toYYYYMM(EventDate) ORDER BY (OrderID, BannerID)
```

Create Table and Fill Some Data

Create Table:

```
CREATE TABLE mt (
    EventDate Date,
    OrderID Int32,
    BannerID UInt64,
    GoalNum Int8
) ENGINE = MergeTree()
PARTITION BY toYYYYMM(EventDate) ORDER BY (OrderID, BannerID)
```

Fill Data (twice):

```
INSERT INTO mt SELECT toDate('2018-09-26'),
    number, number + 10000, number % 128 from numbers(1000000);
INSERT INTO mt SELECT toDate('2018-10-15'),
    number, number + 10000, number % 128 from numbers(1000000, 1000000);
```

Table on Disk

metadata:

```
$ ls /var/lib/clickhouse/metadata/default/
mt.sql
```

Table on Disk

metadata:

```
$ ls /var/lib/clickhouse/metadata/default/
mt.sql
```

data:

```
$ ls /var/lib/clickhouse/data/default/mt
201809_2_2_0  201809_3_3_0  201810_1_1_0  201810_4_4_0 201810_1_4_1
detached  format_version.txt
```

Table on Disk

metadata:

```
$ ls /var/lib/clickhouse/metadata/default/
mt.sql
```

data:

```
$ ls /var/lib/clickhouse/data/default/mt
201809_2_2_0  201809_3_3_0  201810_1_1_0  201810_4_4_0 201810_1_4_1
detached  format_version.txt
```

Contents:

- Format file `format_version.txt`
- Directories with parts
- Directory for detached parts

Parts: Details

- Part of the data in PK order
- Contain interval of insert numbers
- Created for each `INSERT`
- Cannot be changed (immutable)

Parts: Why?

PK ordering is needed for efficient OLAP queries

- In our case `(OrderID, BannerID)`

Parts: Why?

- PK ordering is needed for efficient OLAP queries
  - In our case `(OrderID, BannerID)`
- But the data comes in order of time
  - By `EventDate`

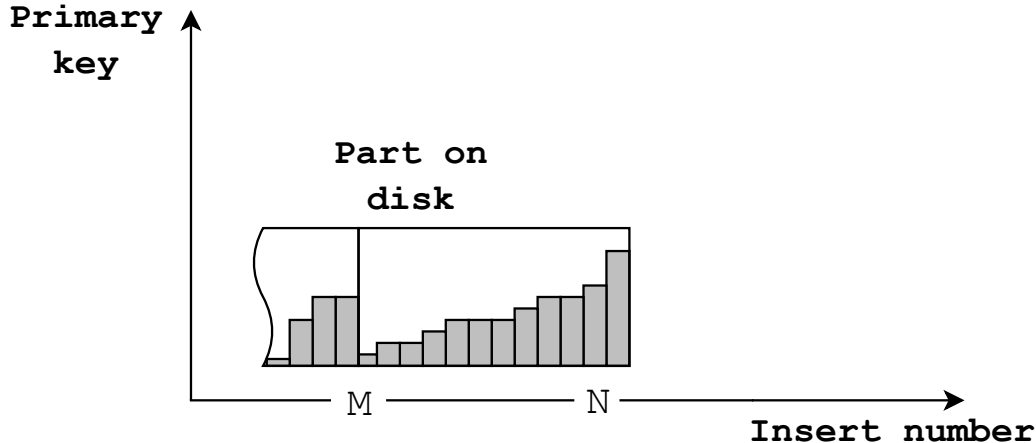
Parts: Why?

- PK ordering is needed for efficient OLAP queries
  - In our case `(OrderID, BannerID)`
- But the data comes in order of time
  - By `EventDate`
- Re-sorting all the data is expensive
  - ClickHouse handle hundreds of terabytes

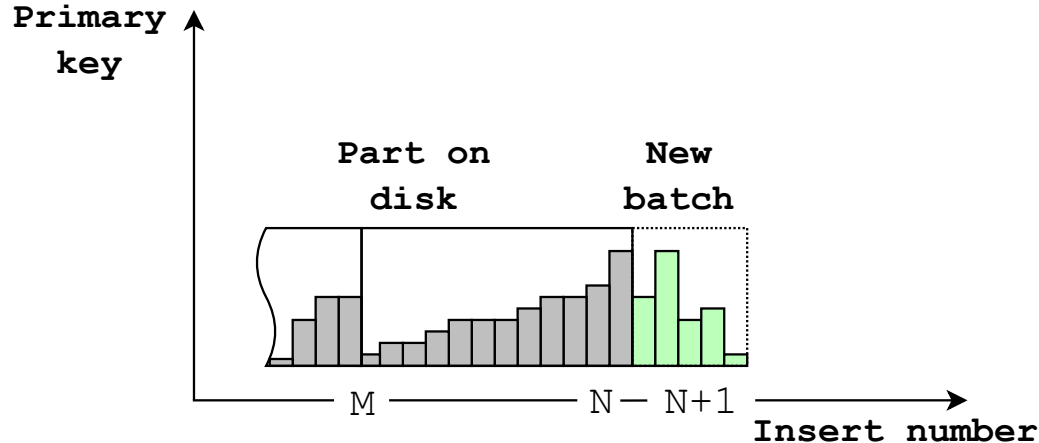
Parts: Why?

- PK ordering is needed for efficient OLAP queries
  - In our case `(OrderID, BannerID)`
- But the data comes in order of time
  - By `EventDate`
- Re-sorting all the data is expensive
  - ClickHouse handle hundreds of terabytes
- **Solution:** Store data in a set of ordered parts!

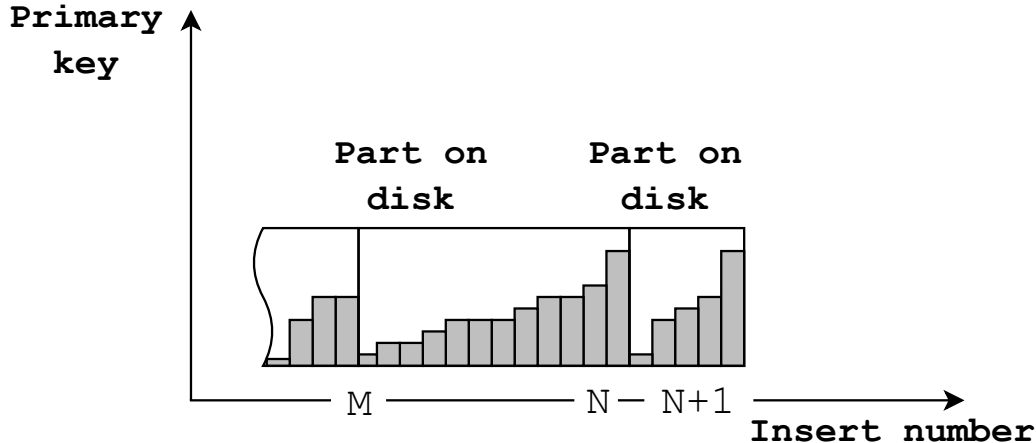
Parts: Main Idea



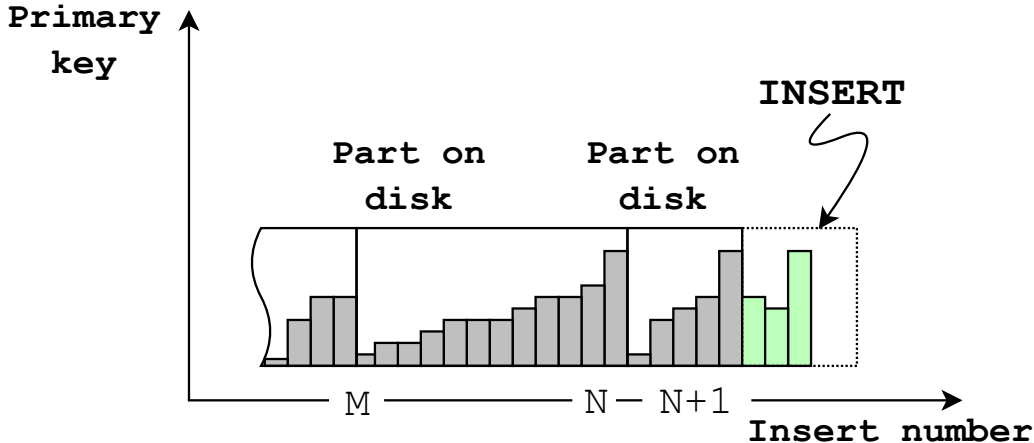
Parts: Main Idea



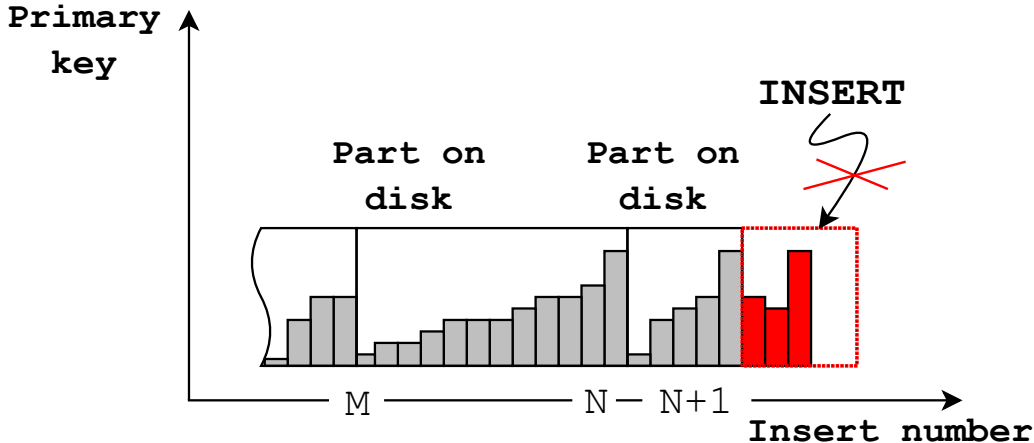
Parts: Main Idea



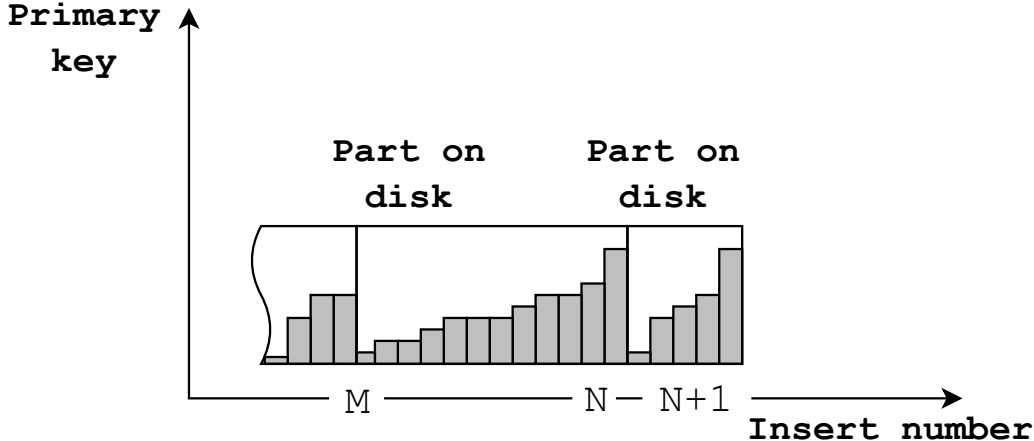
Parts: Atomic insert



Parts: Atomic insert



Parts: Atomic insert



Read

Parts: Data

```
$ ls /var/lib/clickhouse/data/default/mt/201810_1_4_1
GoalNum.bin GoalNum.mrk BannerID.bin ...
primary.idx checksums.txt count.txt columns.txt
partition.dat minmax_EventDate.idx
```

Parts: Data

```
$ ls /var/lib/clickhouse/data/default/mt/201810_1_4_1
GoalNum.bin GoalNum.mrk BannerID.bin ...
primary.idx checksums.txt count.txt columns.txt
partition.dat minmax_EventDate.idx
```

Contents:

- `primary.idx` – primary key on disk

Parts: Data

```
$ ls /var/lib/clickhouse/data/default/mt/201810_1_4_1
GoalNum.bin GoalNum.mrk BannerID.bin ...
primary.idx checksums.txt count.txt columns.txt
partition.dat minmax_EventDate.idx
```

Contents:

- `primary.idx` – primary key on disk
- `GoalNum.bin` – compressed column
- `GoalNum.mrk` – marks for column

Parts: Data

```
$ ls /var/lib/clickhouse/data/default/mt/201810_1_4_1
GoalNum.bin GoalNum.mrk BannerID.bin ...
primary.idx checksums.txt count.txt columns.txt
partition.dat minmax_EventDate.idx
```

Contents:

- `primary.idx` – primary key on disk
- `GoalNum.bin` – compressed column
- `GoalNum.mrk` – marks for column
- `partition.dat` – partition id

Parts: Data

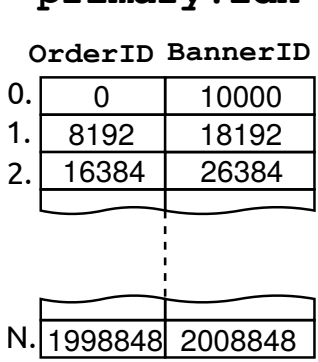
```
$ ls /var/lib/clickhouse/data/default/mt/201810_1_4_1
GoalNum.bin GoalNum.mrk BannerID.bin ...
primary.idx checksums.txt count.txt columns.txt
partition.dat minmax_EventDate.idx
```

Contents:

- `primary.idx` – primary key on disk
- `GoalNum.bin` – compressed column
- `GoalNum.mrk` – marks for column
- `partition.dat` – partition id
- `...` – a lot of other useful files

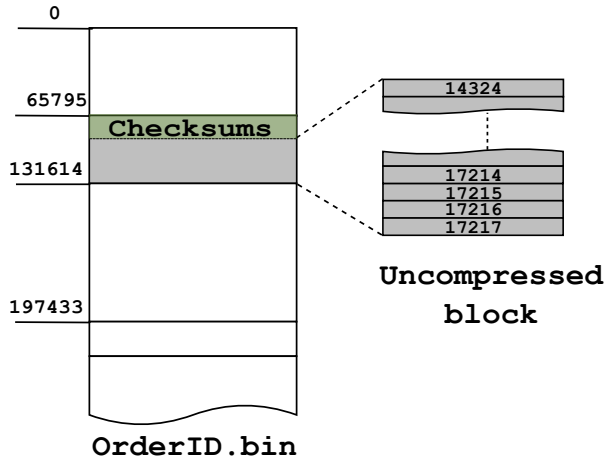
Index

- Row-oriented
- Sparse (each 8192 row)
- Stored in memory
- Uncompressed



Columns

- Each column in separate file
- Compressed by blocks
- Checksums for each block



How to Use Index?

Problem:

- Index is sparse and contain rows
- Columns contain compressed blocks

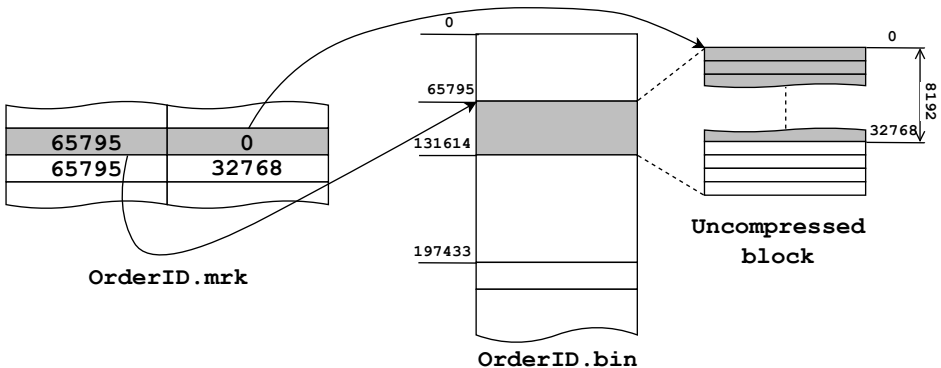
How to match index with columns?

Solution: Marks

- Mark – offset in compressed file and uncompressed block
- Stored in `column_name.mrk` files
- One for each index row

Solution: Marks

- Mark – offset in compressed file and uncompressed block
- Stored in `column_name.mrk` files
- One for each index row



Put it all together

Algorithm:

- Determine required index rows
- Found corresponding marks
- Distribute granules (stripe of marks) among threads
- Read required granules

Properties:

- Read granules concurrently
- Threads can steal tasks

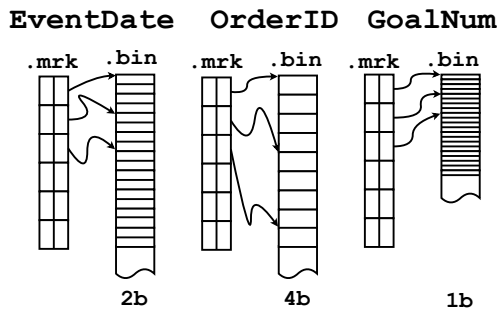
Read Data from Disk

primary.idx

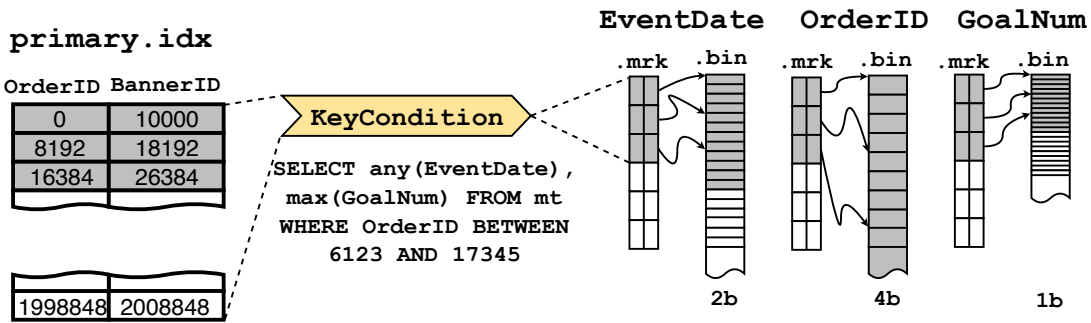
OrderID	BannerID
0	10000
8192	18192
16384	26384
1998848	2008848

KeyCondition

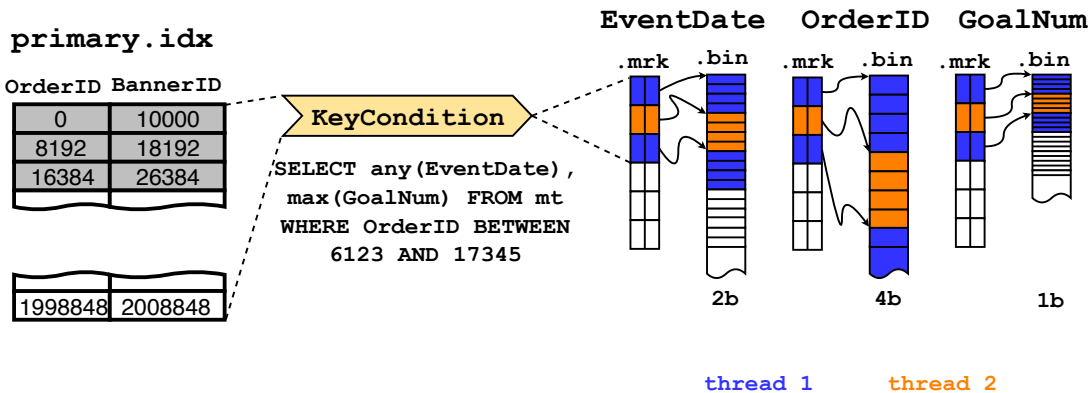
```
SELECT any(EventDate),
max(GoalNum) FROM mt
WHERE OrderID BETWEEN
6123 AND 17345
```



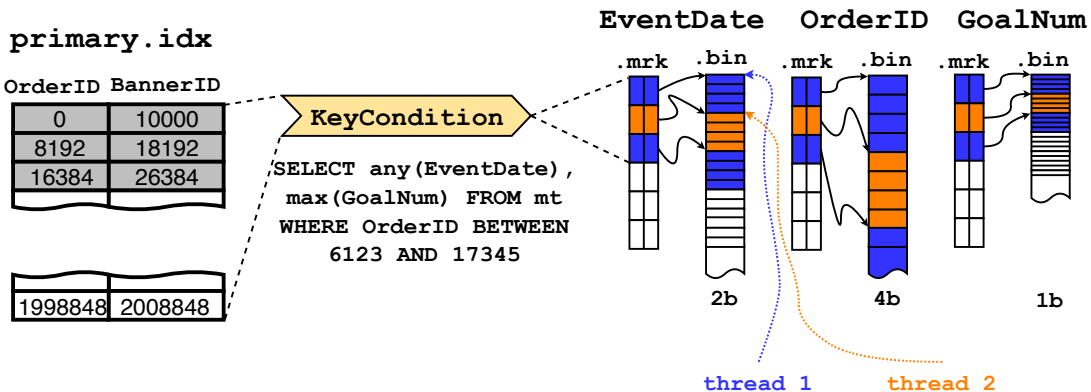
Read Data from Disk



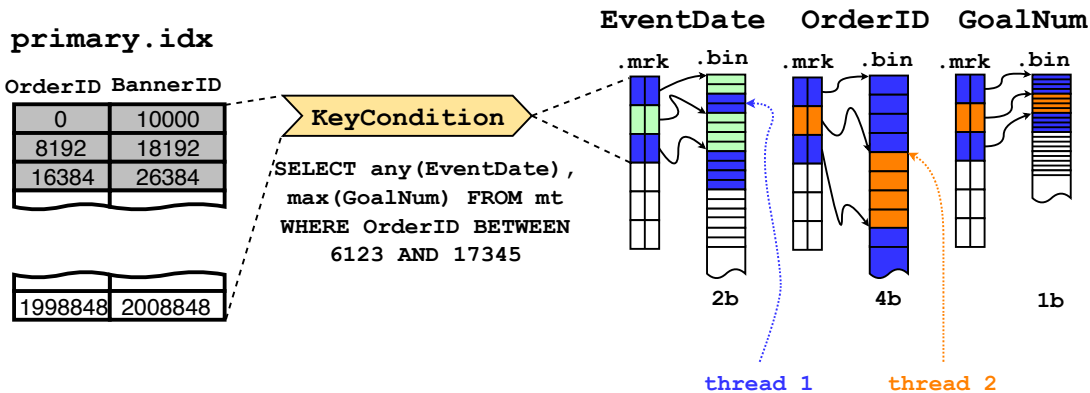
Read Data from Disk



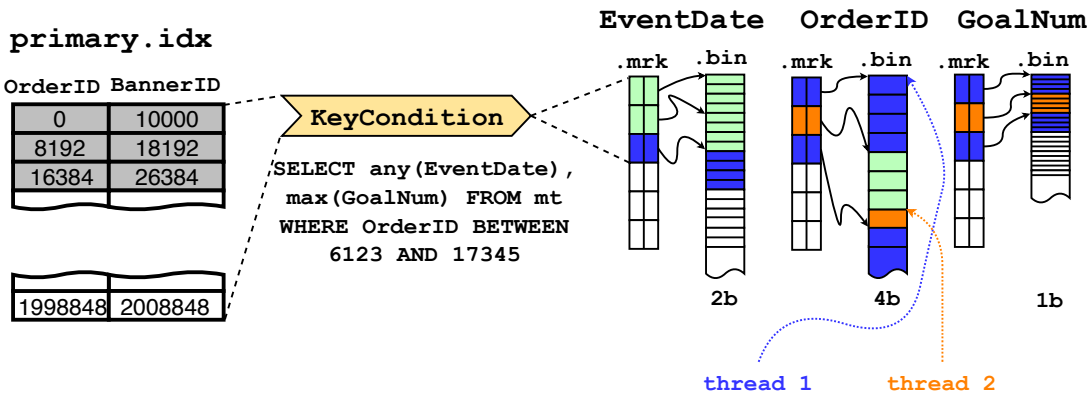
Read Data from Disk



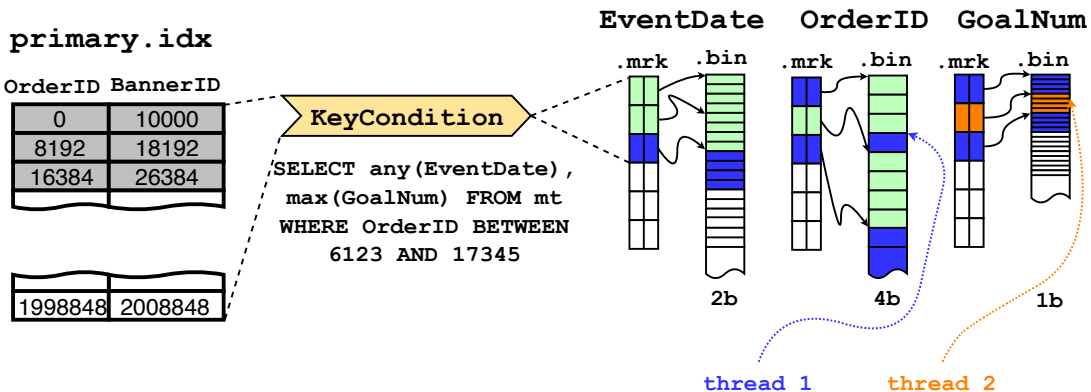
Read Data from Disk



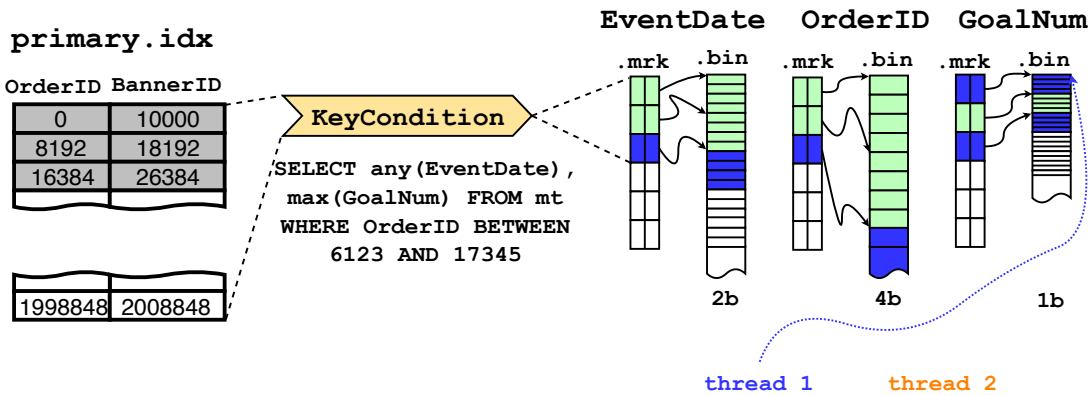
Read Data from Disk



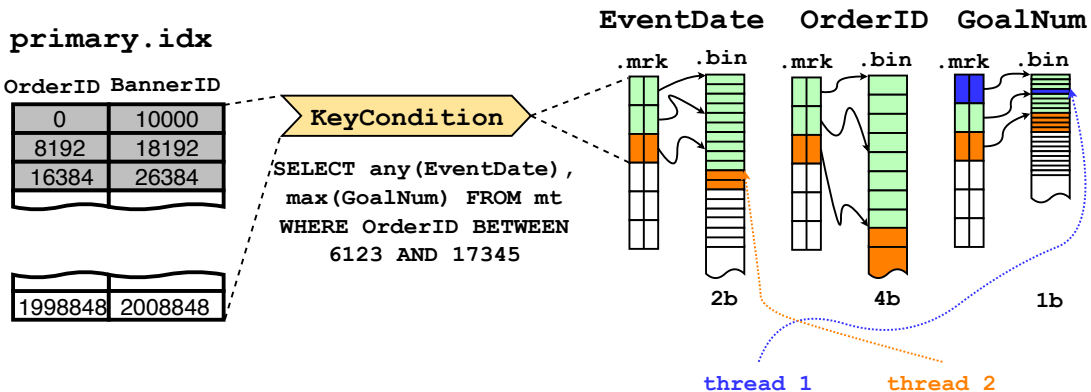
Read Data from Disk



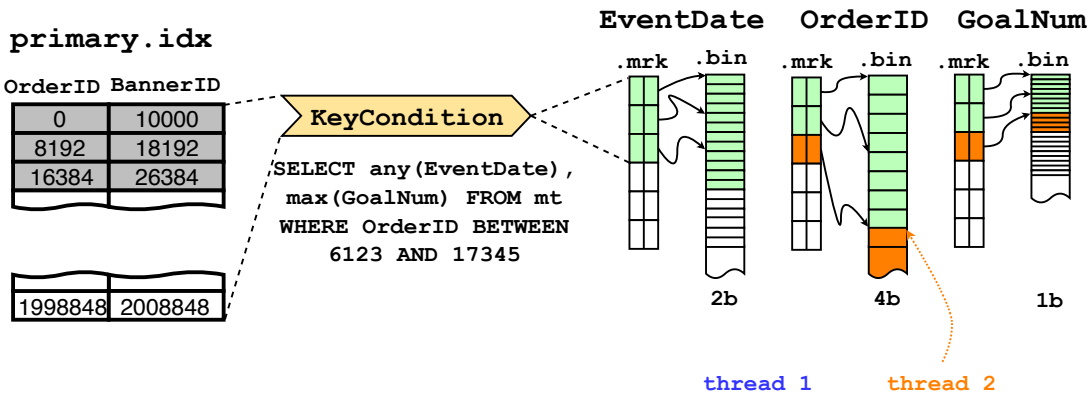
Read Data from Disk



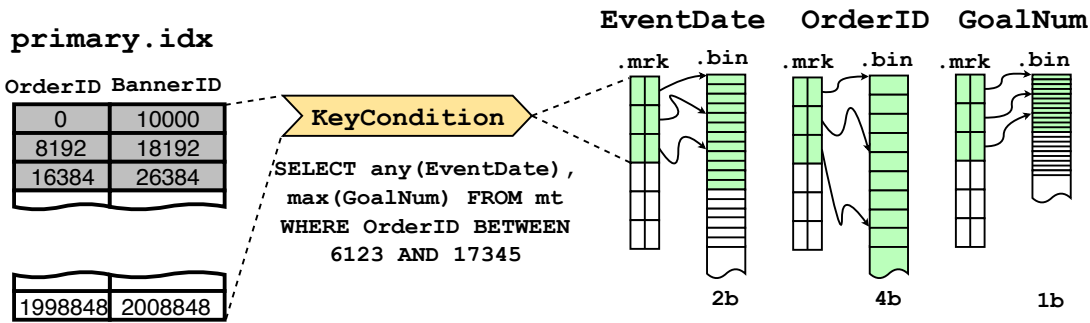
Read Data from Disk



Read Data from Disk



Read Data from Disk



Compaction

Problem: Amount Files in Parts

Test example: Almost OK

```
$ ls /var/lib/clickhouse/data/default/mt/201810_1_4_1 | wc -l
14
```

Problem: Amount Files in Parts

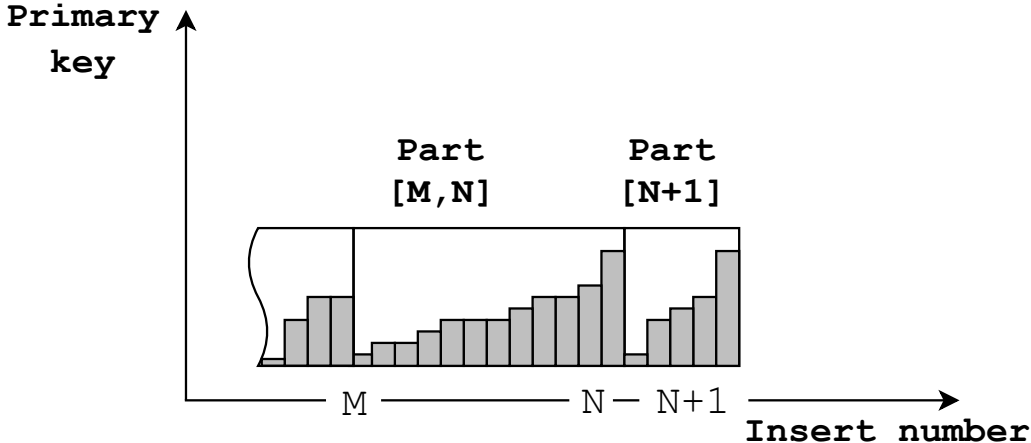
Test example: Almost OK

```
$ ls /var/lib/clickhouse/data/default/mt/201810_1_4_1 | wc -l
14
```

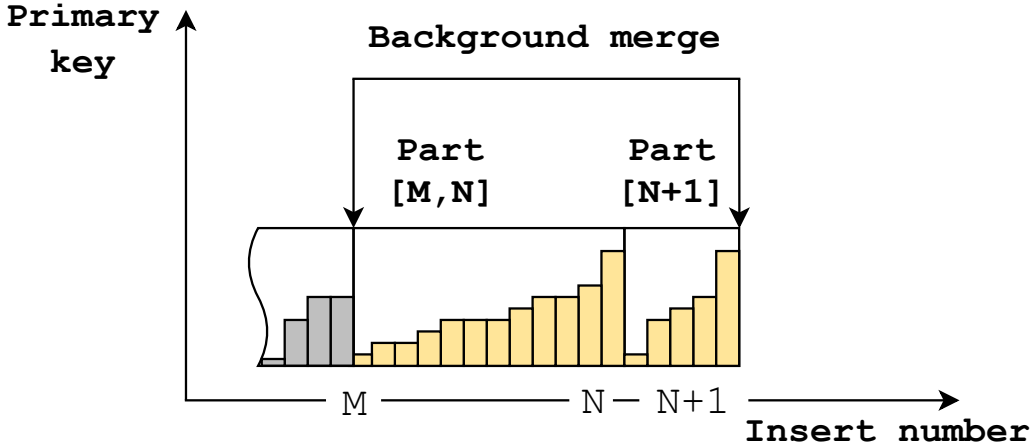
Production: Too much

```
$ ssh -A root@mtgiga075-2.metrika.yandex.net
# ls /var/lib/clickhouse/data/merge/visits_v2/202002_4462_4462_0 | wc -l
1556
```

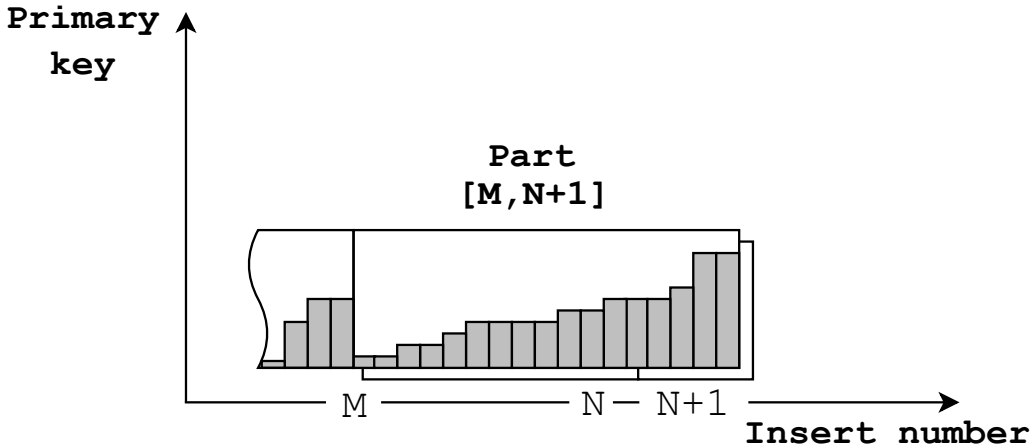
Solution: Merges



Solution: Merges



Solution: Merges



Properties of Merge

- Each part participate in a single successful merge
- Source parts became **inactive**
- Addional logic during merge

Things to do while merging

Replace/update records

- `ReplacingMergeTree` – replace
- `SummingMergeTree` – sum
- `CollapsingMergeTree` – fold
- `VersionedCollapsingMergeTree` – fold rows + versioning

Pre-aggregate data

- `AggregatingMergeTree` – merge aggregate function states

Metrics rollup

- `GraphiteMergeTree` – rollup in graphite fashion

Modify

Partitioning

```
ENGINE = MergeTree() PARTITION BY toYYYYMM(EventDate) ORDER BY ...
```

Partitioning

```
ENGINE = MergeTree() PARTITION BY toYYYYMM(EventDate) ORDER BY ...
```

- Logical entities (doesn't stored on disk)
- Table can be partitioned by any expression (default: by month)
- Parts from different partitions never merged
- `MinMax` index by partition columns
- Easy manipulation of partitions:

Partitioning

```
ENGINE = MergeTree() PARTITION BY toYYYYMM(EventDate) ORDER BY ...
```

- Logical entities (doesn't stored on disk)
- Table can be partitioned by any expression (default: by month)
- Parts from different partitions never merged
- `MinMax` index by partition columns
- Easy manipulation of partitions:

```
ALTER TABLE mt DROP PARTITION 201810
ALTER TABLE mt DETACH/ATTACH PARTITION 201809
```

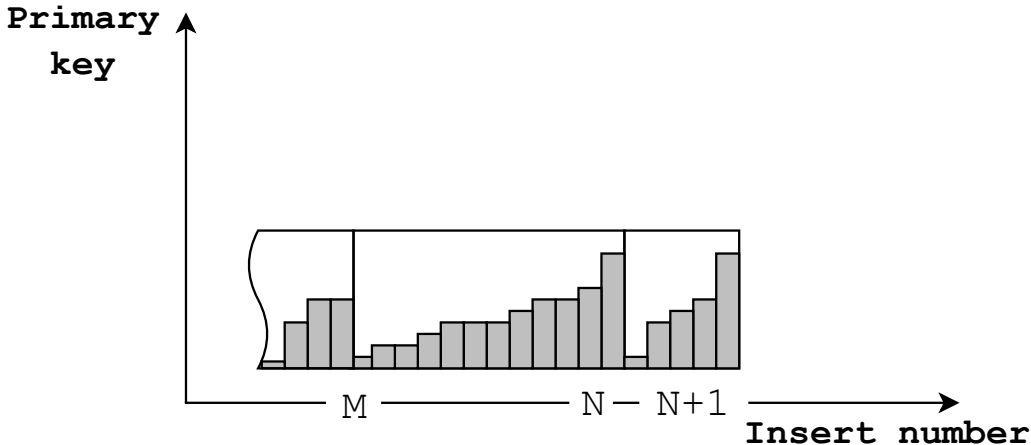
Mutations

```
ALTER TABLE mt DELETE WHERE OrderID < 1205
ALTER TABLE mt UPDATE GoalNum = 3 WHERE BannerID = 235433;
```

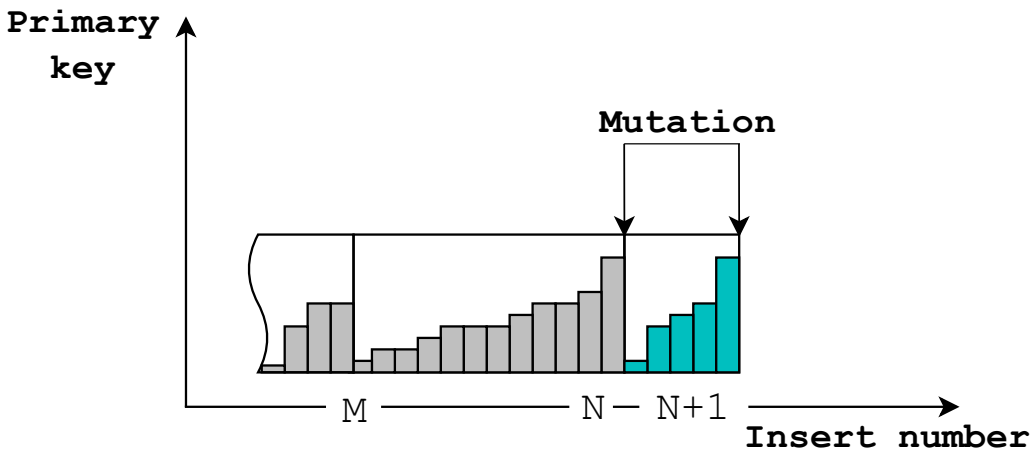
Features

- NOT designed for regular usage
- Overwrite all touched parts on disk
- Work in background
- Original parts became **inactive**

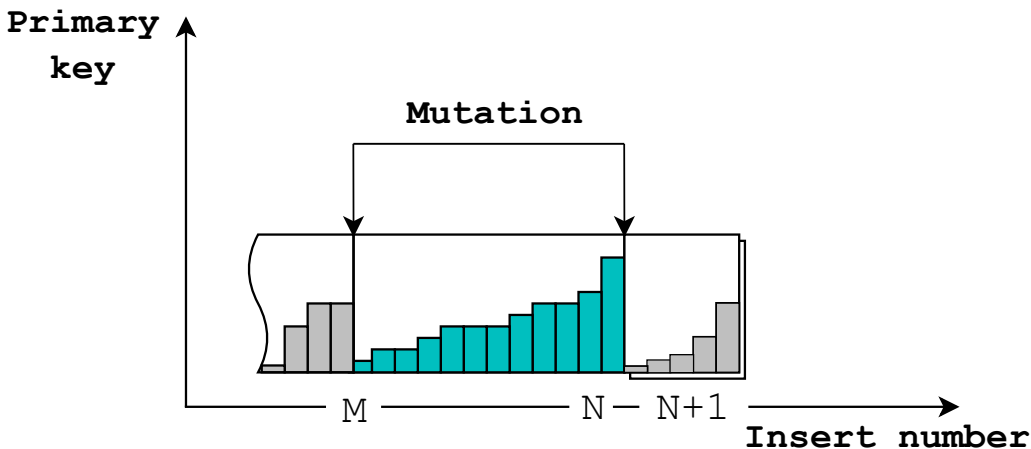
Mutations



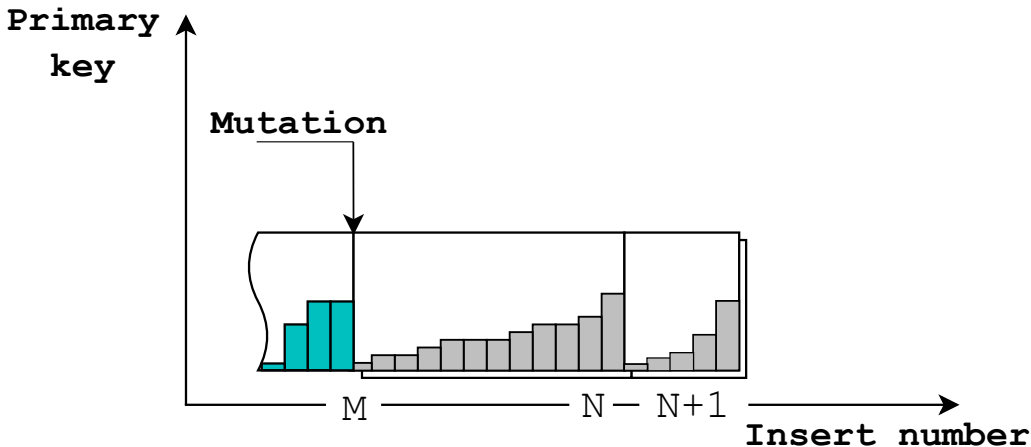
Mutations



Mutations

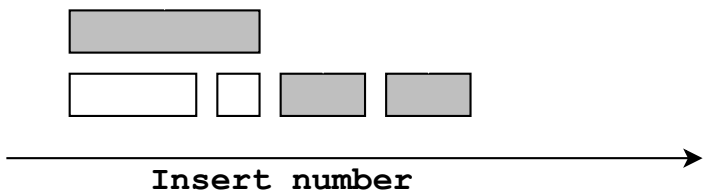


Mutations

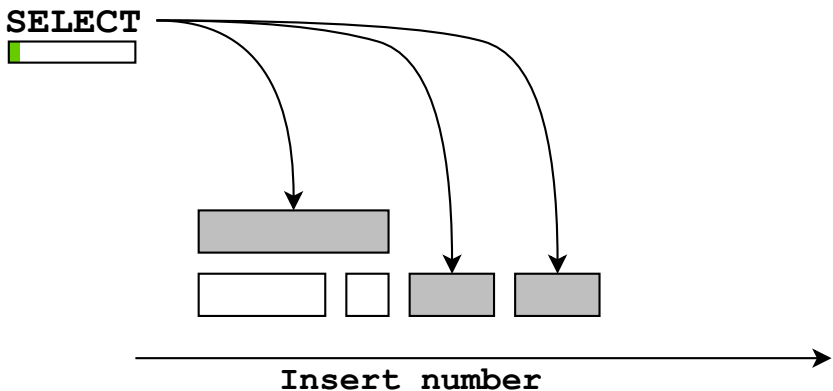


All together

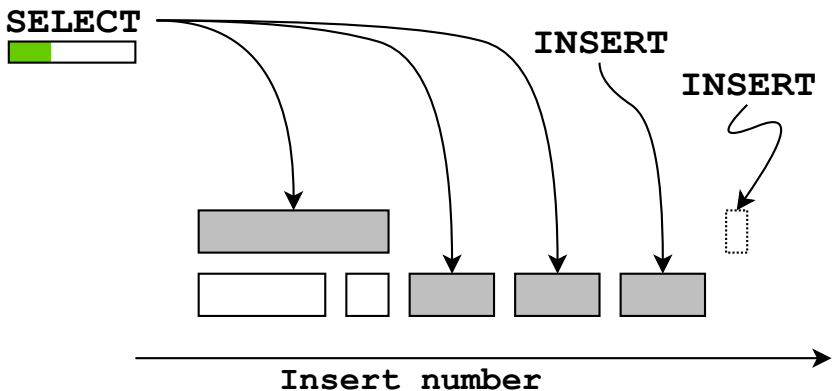
Parts Lifetime



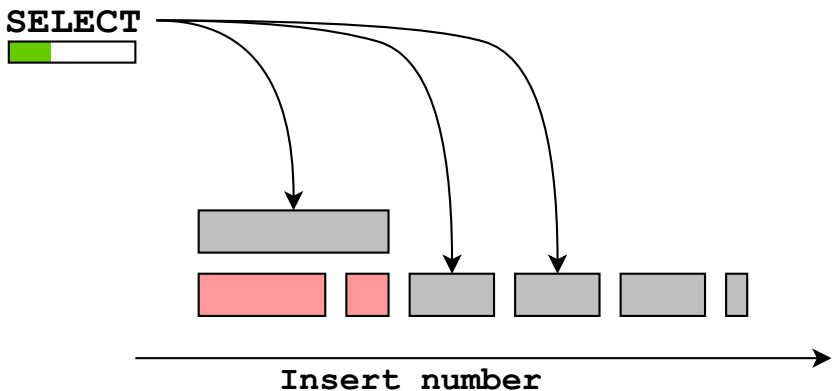
Parts Lifetime



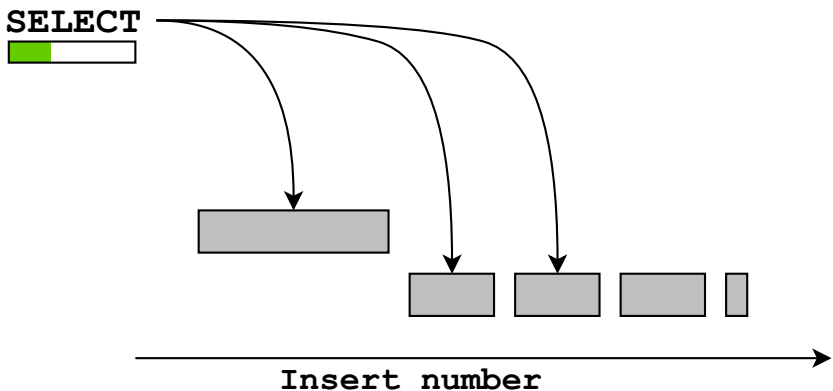
Parts Lifetime



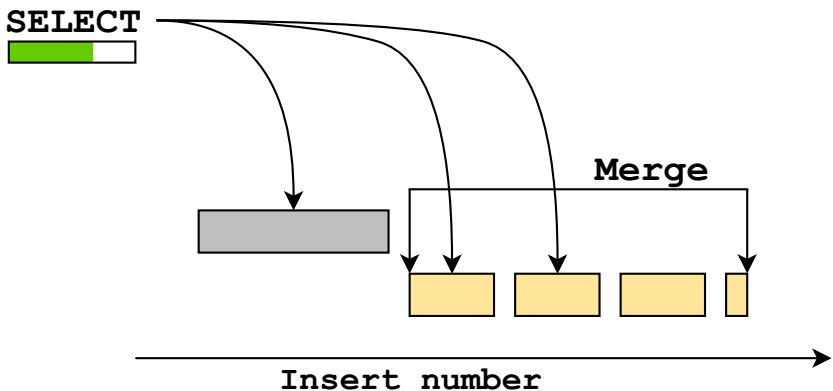
Parts Lifetime



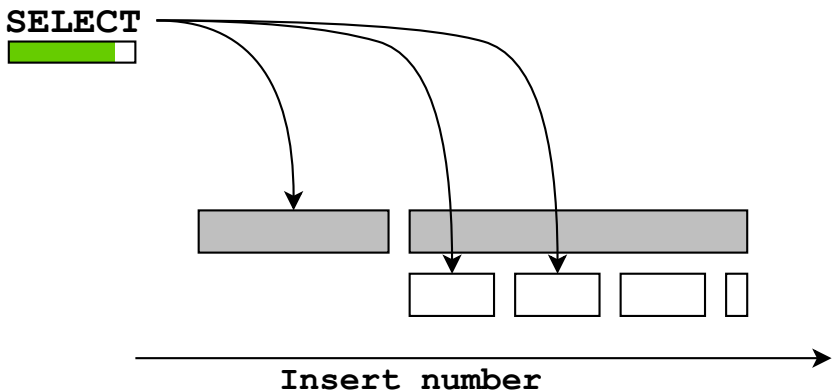
Parts Lifetime



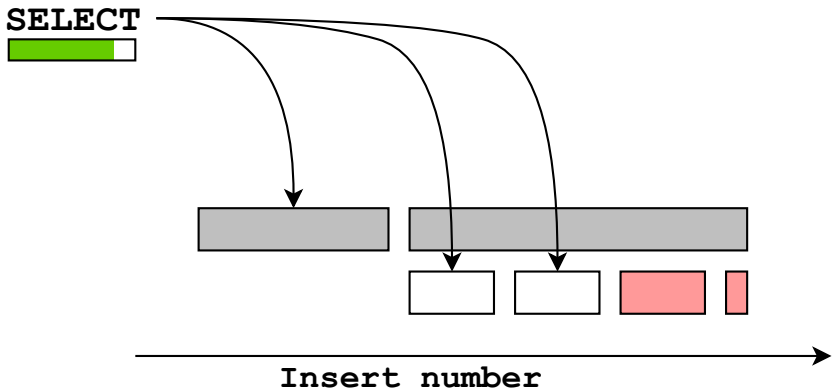
Parts Lifetime



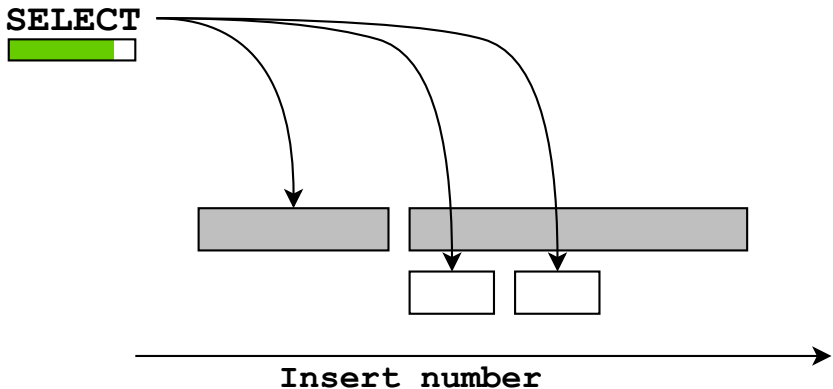
Parts Lifetime



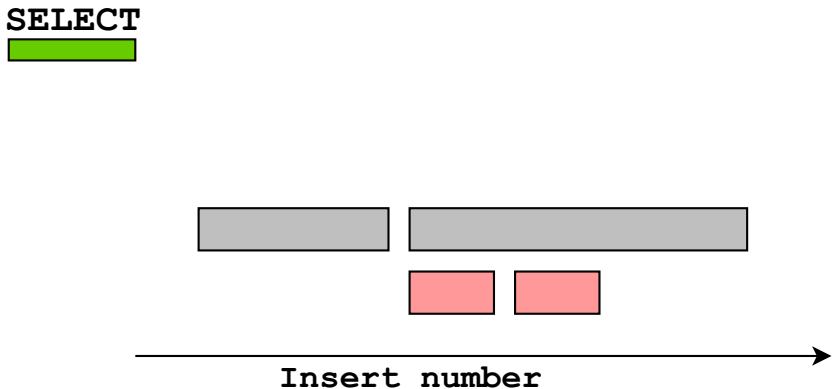
Parts Lifetime



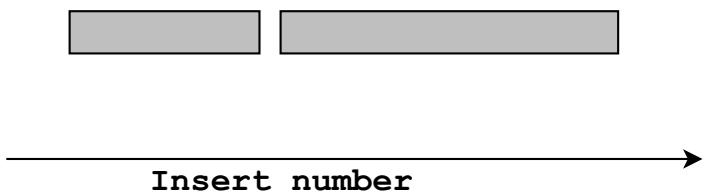
Parts Lifetime



Parts Lifetime

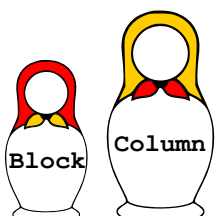


Parts Lifetime

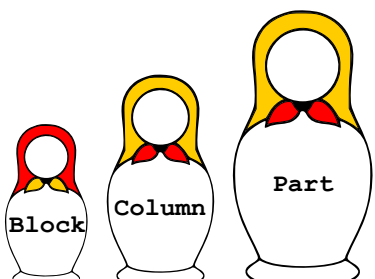


Summarize: MergeTree Consists of

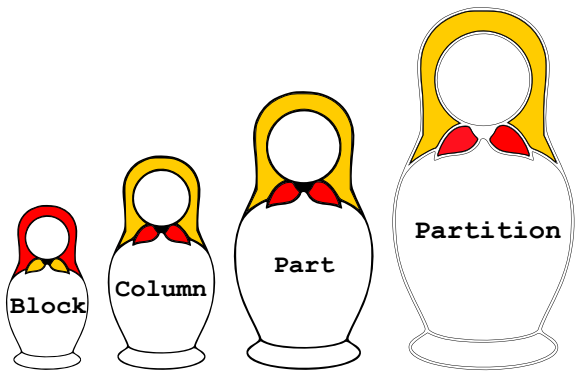
Summarize: MergeTree Consists of



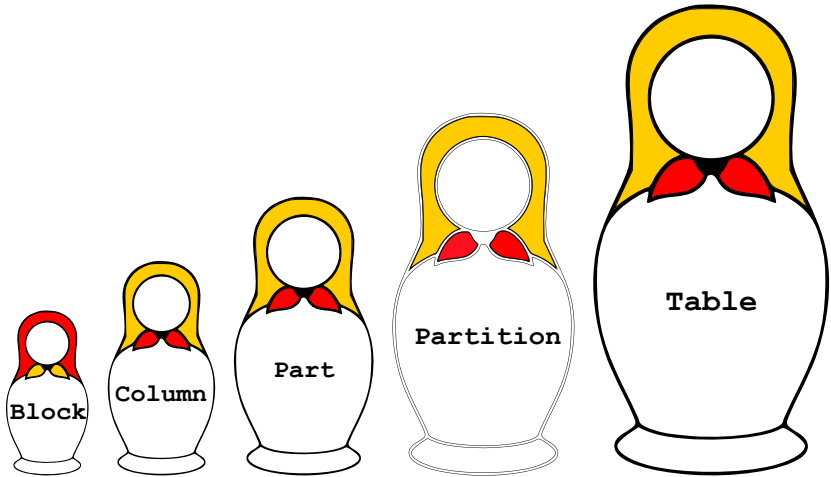
Summarize: MergeTree Consists of



Summarize: MergeTree Consists of



Summarize: MergeTree Consists of



Things to remember

Control total number of parts

- Rate of INSERTs

Things to remember

- **Control** total number of parts
  - Rate of INSERTs
- **Merging** runs in the background
  - Even when there are no queries!
  - With additional fold logic

Things to remember

- **Control** total number of parts
  - Rate of INSERTs
- **Merging** runs in the background
  - Even when there are no queries!
  - With additional fold logic
- **Index** is sparse
  - Must fit into memory
  - Determines order of data on disk
  - Using the index is always beneficial

Things to remember

- **Control** total number of parts
  - Rate of INSERTs
- **Merging** runs in the background
  - Even when there are no queries!
  - With additional fold logic
- **Index** is sparse
  - Must fit into memory
  - Determines order of data on disk
  - Using the index is always beneficial
- **Partitions** is logical entity
  - Easy manipulation with portions of data
  - Cannot improve SELECTs performance

Thank you

QA