PERCONA LIVE ONLINE MAY 12 - 13th 2021

Projections in ClickHouse

Amos Bird (郑天祺) , Software Engineer at KuaiShou

Ph.D in Institute of Computing Technology, Chinese Academy of Sciences

About Me

- Active ClickHouse Contributor
  1. ~300 merged PRs
  2. ~40 Stack Overflow Answers
  3. Doing some code reviews occasionally
  4. Helping new ClickHouse developers

- Graduated from ICT CAS with a Ph.D degree in database

- Currently at KuaiShou Data Platform Department

https://github.com/amosbird

Agenda

- Short intro about ***Projection***
- Demo of using ***ClickHouse Projection***
- How we implement ***Projection*** in ClickHouse
- Pros & Cons and some experiments
- Future works

What is “*Projection*”?

From C-Store paper: “

*Hence, C-Store physically stores a collection of columns, each sorted on some attribute(s). Groups of columns sorted on the same attribute are referred to as "**projections**"; the same column may exist in multiple projections, possibly sorted on a different attribute in each.*

What is “Projection”?

- Originated from C-Store/Vertica (Don't confuse it with SQL’s Projection operation)
  1. Projections are collections of table columns
  2. Projections store data in a format that optimizes query execution
  3. Projections can store derived data to optimize various kind of queries, e.g. aggregations

- ClickHouse Projection
  1. Defined by tailored SELECT query
  2. Support arbitrary functions and their arbitrary combinations
  3. Can be used physically after materialization, or logically as a view, or mixed

What Problems to Solve?

ClickHouse stores data in LSM-like format (MergeTree Family)

1. Can only have one ordering of columns
    a. ORDER BY (*author_id, photo_id*), what if we need to query with *photo_id* alone?
    b. Z-Curve index is still under-development, and has its own problem
    c. Skip-index works badly when data is scattered in many granules

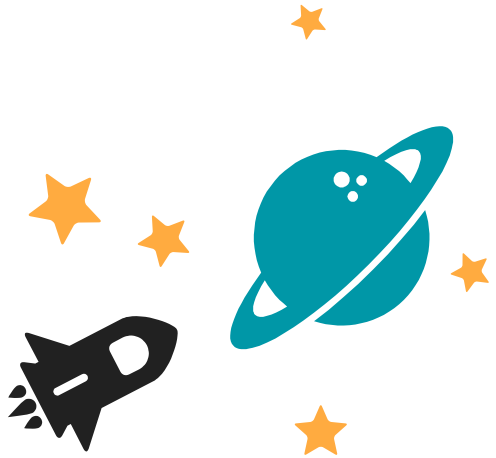
2. Pre-aggregation requires manual construction
    a. AggregatingMergeTree can only aggregate in one way
    b. Queries need manual rewriting to do aggregation over pre-aggregation
    c. It’s not possible to query detail data anymore

Projections to the Rescue

- Projection can reorder columns to benefit queries with various column filters.
- Projection can pre-aggregate columns which reduces both computation and IO.
- Projection query analysis select the projection with least data to scan without modifying user query.
- Projection is defined by SQL in intuitive way: GROUP BY means pre-aggregating and ORDER BY means reordering.
- Projection stores data as MergeTree data parts in both cases with strong consistency guarantee. Detail data is kept and can be used to answer queries along with projections.

Two Birds, One Stone!

A Demo of *ClickHouse Projection*



ClickHouse Projection Demo

Let’s create a table to demonstrate the power of projection.

```
CREATE TABLE video_log
(
    `datetime` DateTime, -- 20,000 records per second
    `user_id` UInt64, -- Cardinality == 100,000,000
    `device_id` UInt64, -- Cardinality == 200,000,000
    `domain` LowCardinality(String), -- Cardinality == 100
    `bytes` UInt64, -- Ranging from 128 to 1152
    `duration` UInt64 -- Ranging from 100 to 400
)
ENGINE = MergeTree
PARTITION BY toDate(datetime) -- Daily partitioning
ORDER BY (user_id, device_id); -- Can only favor one column here
```

ClickHouse Projection Demo

Let’s populate the table with one day’s data using a ***GenerateRandom*** table.

```
CREATE TABLE rng
(
    `user_id_raw` UInt64,
    `device_id_raw` UInt64,
    `domain_raw` UInt64,
    `bytes_raw` UInt64,
    `duration_raw` UInt64
)
ENGINE = GenerateRandom(1024);
```

```
INSERT INTO video_log SELECT
    toUnixTimestamp(toDateTime(today()))
       + (rowNumberInAllBlocks() / 20000),
    user_id_raw % 100000000 AS user_id,
    device_id_raw % 200000000 AS device_id,
    domain_raw % 100,
    (bytes_raw % 1024) + 128,
    (duration_raw % 300) + 100
FROM rng
LIMIT 1728000000;
```

ClickHouse Projection Demo

Case 1: Finding the hourly video stream property of a given user today.

```
SELECT
    toStartOfHour(datetime) AS hour,
    sum(bytes),
    avg(duration)
FROM video_log
WHERE (toDate(hour) = today()) AND (user_id = 100)
GROUP BY hour;
```

19 rows in set. Elapsed: 0.017 sec. Processed 32.77 thousand rows,

Log: 4/210940 marks by primary key, 4 marks to read from 4 ranges

ClickHouse Projection Demo

Case 2: Finding the hourly video stream property of a given device today.

```
SELECT
    toStartOfHour(datetime) AS hour,
    sum(bytes),
    avg(duration)
FROM video_log
WHERE (toDate(hour) = today()) AND (device_id = '100')
GROUP BY hour;
```

7 rows in set. Elapsed: 8.434 sec. Processed 1.73 billion rows,

Log: 210940/210940 marks by primary key, 210940 marks to read from 4 ranges

ClickHouse Projection Demo

Case 2: Finding the hourly video stream property of a given device today.

Let's add a *normal projection* : *p_norm* to speed up querying by *device_id.*

```
ALTER TABLE video_log ADD PROJECTION p_norm
    (
        SELECT
            datetime,
            device_id,
            bytes,
            duration
        ORDER BY device_id
    );

ALTER TABLE video_log MATERIALIZE PROJECTION p_norm;
```

ClickHouse Projection Demo

Case 2: Finding the hourly video stream property of a given device today.

Try again with the same query.

```
SELECT
    toStartOfHour(datetime) AS hour,
    sum(bytes),
    avg(duration)
FROM video_log
WHERE (toDate(hour) = today()) AND (device_id = '100')
GROUP BY hour;
```

7 rows in set. Elapsed: 0.055 sec. Processed 24.58 thousand rows,

153x faster!!

Log: 3/210940 marks by primary key, 3 marks to read from 3 ranges

ClickHouse Projection Demo

Case 3: Finding the hourly video stream property aggregated by domain today.

```
SELECT
    toStartOfHour(datetime) AS hour,
    domain,
    sum(bytes),
    avg(duration)
FROM video_log
WHERE toDate(hour) = today()
GROUP BY hour, domain;
```

2400 rows in set. Elapsed: 11.493 sec. Processed 1.73 billion rows,

Log: 210940/210940 marks by primary key, 210940 marks to read from 4 ranges
Aggregate 1728000000 rows to 2400 rows

ClickHouse Projection Demo

Case 3: Finding the hourly video stream property aggregated by domain today.

Let's add an *aggregate projection*: *p_agg* to speed up grouping by *domain*.

```
ALTER TABLE video_log ADD PROJECTION p_agg
    (
        SELECT
            toStartOfHour(datetime) AS hour,
            domain,
            sum(bytes),
            avg(duration)
        GROUP BY
            hour,
            domain
    );

ALTER TABLE video_log MATERIALIZE PROJECTION p_agg;
```

Aggregate projections are defined by the to-be-accelerated query!

ClickHouse Projection Demo

Case 3: Finding the hourly video stream property aggregated by domain today.

Try again with the same query.

```
SELECT
    toStartOfHour(datetime) AS hour,
    domain,
    sum(bytes),
    avg(duration)
FROM video_log
WHERE toDate(hour) = today()
GROUP BY hour, domain;
```

2400 rows in set. Elapsed: 0.029 sec. Processed 5.20 thousand rows,

396x faster!!

Log: 4/4 marks by primary key, 4 marks to read from 4 ranges
Aggregate 5200 rows to 2400 rows

ClickHouse Projection Demo

What about space consumption? Let's find it out via *system.projection_parts.*

Normal projection: *p_norm*

```
SELECT
    name,
    parent_name,
    formatReadableSize(bytes_on_disk) AS bytes,
    formatReadableSize(parent_bytes_on_disk) AS parent_bytes,
    bytes_on_disk / parent_bytes_on_disk AS ratio
FROM system.projection_parts
WHERE (name = 'p_norm') AND (table = 'video_log')
```

```
|--------+--------------------------+-----------+--------------+---------|
| name   | parent_name              | bytes     | parent_bytes |   ratio |
|--------+--------------------------+-----------+--------------+---------|
| p_norm | 20210506_1_740_4_1651    | 8.77 GiB  | 23.94  GiB   | 0.36642 |
| p_norm | 20210506_741_1480_4_1651 | 8.81 GiB  | 24.01  GiB   | 0.36681 |
| p_norm | 20210506_1481_1647_3_1651| 2.09 GiB  | 5.67  GiB    | 0.36895 |
| p_norm | 20210506_1648_1648_0_1651| 14.99 MiB | 38.38  MiB   | 0.39063 |
|--------+--------------------------+-----------+--------------+---------|
```

ClickHouse Projection Demo

What about space consumption? Let's find it out via *system.projection_parts.*

Aggregate Projection: *p_agg*

```
SELECT
    name,
    parent_name,
    formatReadableSize(bytes_on_disk) AS bytes,
    formatReadableSize(parent_bytes_on_disk) AS parent_bytes,
    rows,
    parent_rows,
    rows / parent_rows AS ratio
FROM system.projection_parts
WHERE (name = 'p_agg') AND (table = 'video_log')
```

```
|-------+--------------------------+----------+--------------+------+-------------+-----------|
| name  | parent_name              | bytes    | parent_bytes | rows | parent_rows |     ratio |
|-------+--------------------------+----------+--------------+------+-------------+-----------|
| p_agg | 20210506_1_740_4_1651    | 14.80 KiB | 23.94 GiB   | 1100 |   775923300 | 0.0000014 |
| p_agg | 20210506_741_1480_4_1651 | 16.09 KiB | 24.01 GiB   | 1200 |   775923300 | 0.0000015 |
| p_agg | 20210506_1481_1647_3_1651 | 4.78 KiB | 5.67 GiB    |  300 |   175107015 | 0.0000017 |
| p_agg | 20210506_1648_1648_0_1651 | 26.80 KiB | 38.38 MiB  | 2600 |     1046385 | 0.0024847 |
|-------+--------------------------+----------+--------------+------+-------------+-----------|
```

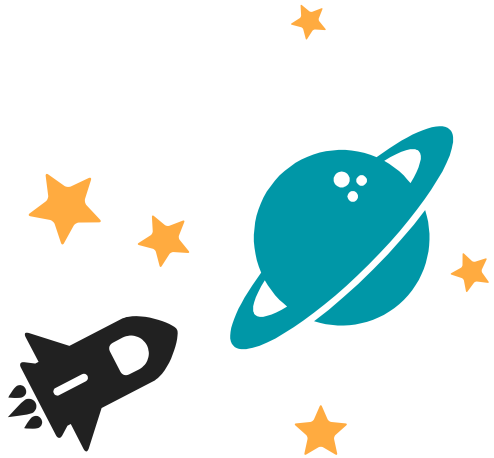
Projection DDL

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
       name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1] [compression_codec] [TTL expr1],
       ...
       PROJECTION projection_name_1 (SELECT <COLUMN LIST EXPR> [GROUP BY] [ORDER BY]),
       ...
) ENGINE = <table_engine> ...
ALTER TABLE [db.]table ADD PROJECTION name (SELECT <COLUMN LIST EXPR> [GROUP BY] [ORDER BY]);
ALTER TABLE [db.]table DROP PROJECTION name;
ALTER TABLE [db.]table MATERIALIZE PROJECTION name [IN PARTITION partition_name];
ALTER TABLE [db.]table CLEAR PROJECTION name [IN PARTITION partition_name];
```

Takeaways from the Demo

- ClickHouse Projections have two types, namely *normal* and *aggregate*.
- Newly added projections only affect newly inserted data.
- In order to build projections for existing data, materialization is required.
- Queries can use projections without any modification.
- Queries can be accelerated even when some data parts don’t have materialized projections.
- Multiple projections can be added and the best one will be selected.

How does it work?



Main constructs of *ClickHouse Projection*

- Projection Definition
  a. Defined by query in an intuitive way
  b. Types are automatically inferred
- Projection Storage
  a. Live inside parent parts
  b. Reuse and follow parent parts when necessary
- Query Analysis
  a. Backtracking query pipeline
  b. Expression name matching
  c. Early index analysis with caching
  d. Query pipeline reconstruction

Projection Definition

- Can be viewed as *CREATE TABLE AS SELECT (CTAS)*
  a. Aliases in SELECT will be expanded and all expressions will use canonical names.

Projection Definition

- Can be viewed as *CREATE TABLE AS SELECT (CTAS)*
  a. Aliases in SELECT will be expanded and all expressions will use canonical names.
  b. *GROUP BY* clause generates *aggregate projections* and uses *AggregatingMergeTree* with given keys.
  c. Aggregate functions generate intermediate data types: *AggregateFunction(...)*. In this case: ***AggregateFunction(sum, UInt64), AggregateFunction(duration, UInt64)***

Projection Definition

- Can be viewed as *CREATE TABLE AS SELECT (CTAS)*
  a. Aliases in SELECT will be expanded and all expressions will use canonical names.
  b. *GROUP BY* clause generates *aggregate projections* and uses *AggregatingMergeTree* with given keys.
  c. Aggregate functions generate intermediate data types: *AggregateFunction(...)*. In this case: ***AggregateFunction(sum, UInt64), AggregateFunction(duration, UInt64)***

```
ALTER TABLE video_log ADD PROJECTION p_agg
    (
        SELECT
            toStartOfHour(datetime) AS hour,
            domain,
            sum(bytes),
            avg(duration)

        GROUP BY
            hour,
            domain
    );
```

Run till the stage before aggregation

```
CREATE TABLE p_agg
ENGINE AggregatingMergeTree
ORDER BY (`toStartOfHour(datetime)`, domain)
AS
    SELECT
        toStartOfHour(datetime),
        domain,
        sum(bytes),
        avg(duration)
    FROM
        video_log
    GROUP BY
        toStartOfHour(datetime), domain
```

Projection Storage

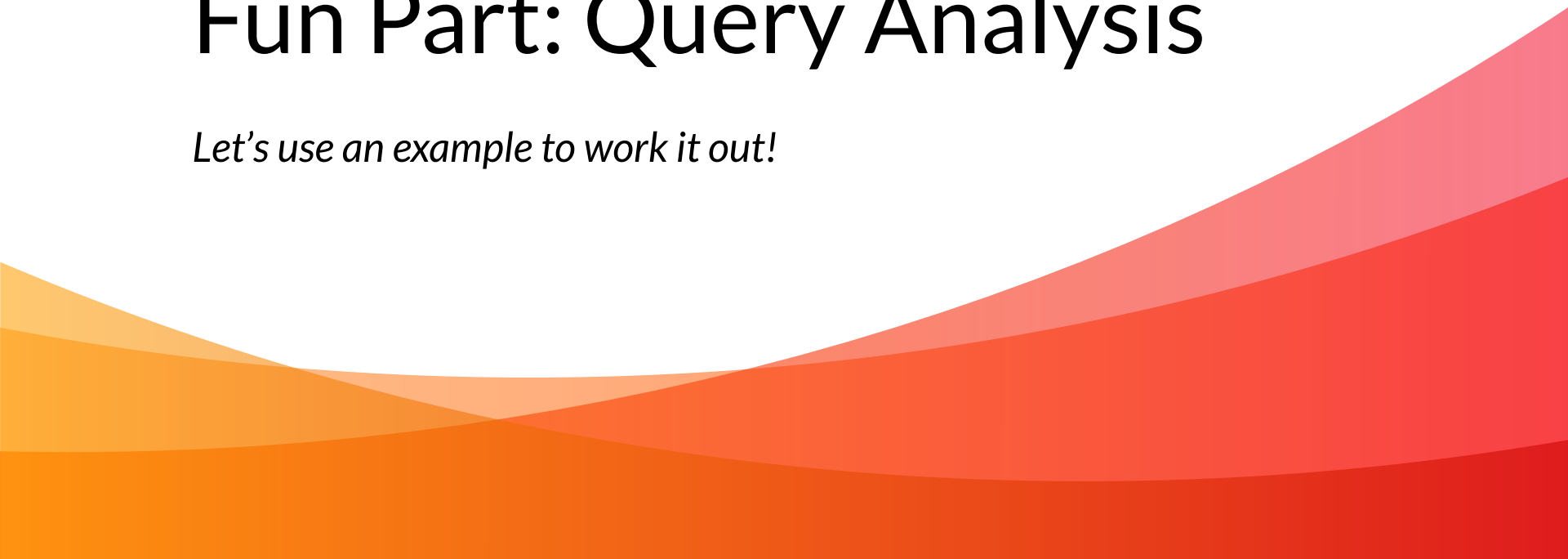
- Projections are parts inside parts
  a. Projection part storage is exactly the same as ordinary *MergeTree* tables.
  b. Projection part uses partition info in parent part directly.
  c. Merge, mutation and replication all follow parent parts.

```
data/data/default/video_log/20210506_1_740_4_1651 (parent_part)
|-- ...
|-- minmax_datetime.idx
|-- partition.dat
|------------------------------------------------- p_agg.proj
`-- p_norm.proj                                    |-- avg%28duration%29.bin
    |-- bytes.bin                                  |-- avg%28duration%29.mrk2
    |-- bytes.mrk2                                 |-- checksums.txt
    |-- checksums.txt                              |-- columns.txt
    |-- columns.txt                                |-- count.txt
    |-- ...                                        |-- ...
    |-- duration.bin                               |-- sum%28bytes%29.bin
    |-- duration.mrk2                              |-- sum%28bytes%29.mrk2
    `-- primary.idx                                |-- toStartOfHour%28datetime%29.bin
                                                   `-- toStartOfHour%28datetime%29.mrk2
```

Shared with all projections

Fun Part: Query Analysis

Let's use an example to work it out!



Query Analysis (Matching Aggregate Projection)

Given Query:

```
SELECT
    toStartOfHour(datetime) AS hour,
    domain,
    sum(bytes),
    avg(duration)
FROM video_log
WHERE toDate(hour) = today()
GROUP BY hour, domain;
```

Table *video_log*:

```
Columns:

  datetime    DateTime,
  user_id     UInt64,
  device_id   String,
  domain      LowCardinality(String),
  bytes       UInt64,
  duration    UInt64,
```

Projection *p_agg:*

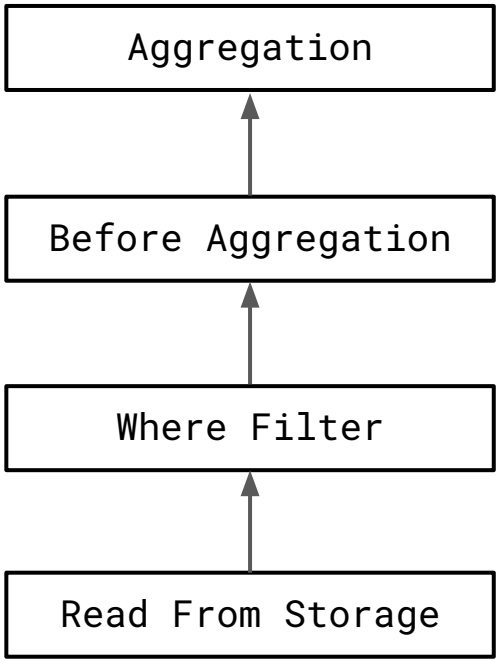
```
Columns:

  toStartOfHour(datetime) DateTime,
  domain LowCardinality(String),
  sum(bytes) AggregateFunction(sum, UInt64),
  avg(duration) AggregateFunction(avg, UInt64),


Primary Keys:
  toStartOfHour(datetime), domain
```

Query Analysis (Matching Aggregate Projection)

Given Query:

```
SELECT
    toStartOfHour(datetime) AS hour,
    domain,
    sum(bytes),
    avg(duration)
FROM video_log
WHERE toDate(hour) = today()
GROUP BY hour, domain;
```

Query Pipeline:

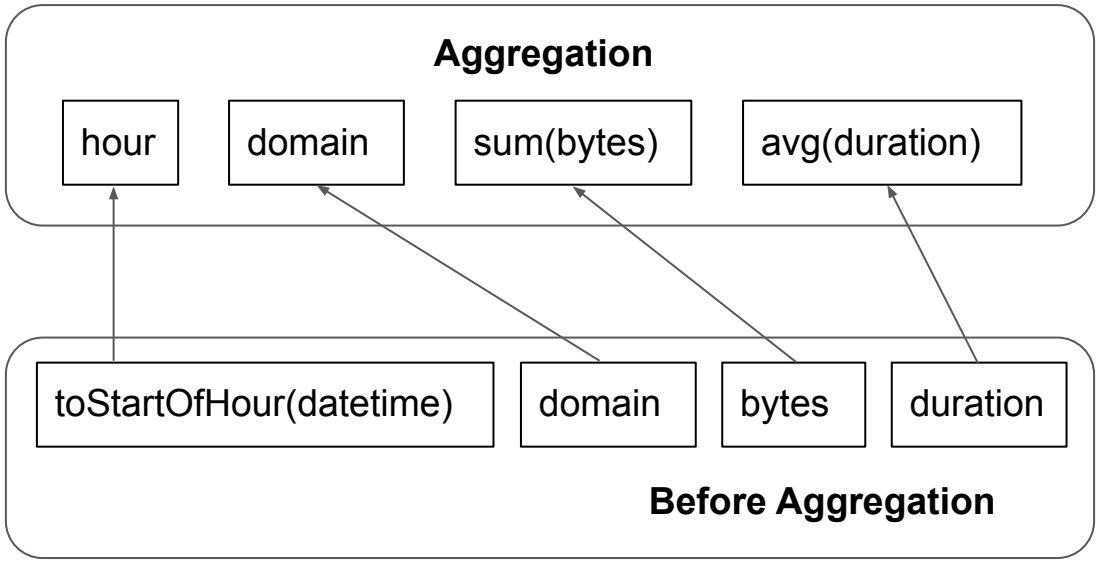


Query Analysis (Matching Aggregate Projection)

Given Query:

```
SELECT
    toStartOfHour(datetime) AS hour,
    domain,
    sum(bytes),
    avg(duration)
FROM video_log
WHERE toDate(hour) = today()
GROUP BY hour, domain;
```

We need the following *action* to run aggregation.

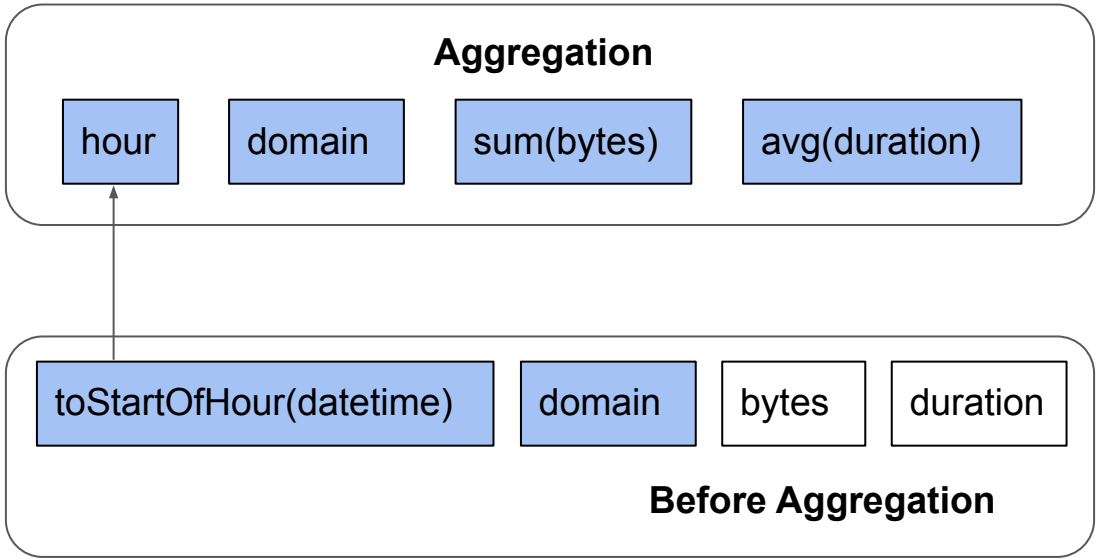


Query Analysis (Matching Aggregate Projection)

Projection *p_agg:*

```
Columns:

  toStartOfHour(datetime),
  domain,
  sum(bytes),
  avg(duration),


Primary Keys:
  toStartOfHour(datetime), domain
```

We found that projection *p_agg* provides all columns needed by this aggregation. **Potential Match!**

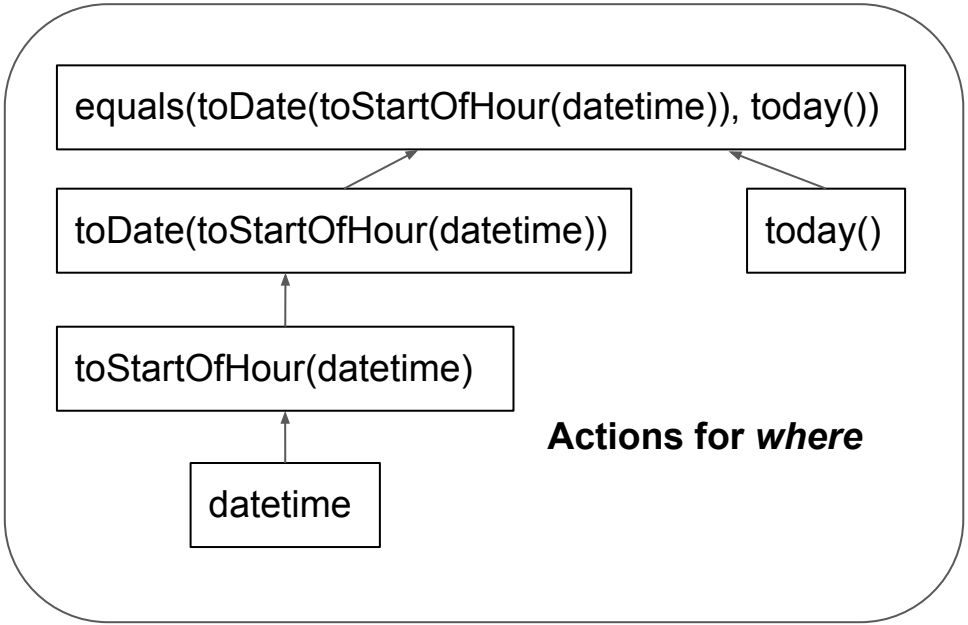


Query Analysis (Matching Aggregate Projection)

Given Query:

```
SELECT
    toStartOfHour(datetime) AS hour,
    domain,
    sum(bytes),
    avg(duration)
FROM video_log
WHERE toDate(hour) = today()
GROUP BY hour, domain;
```

What about *actions* before aggregation like *where*?

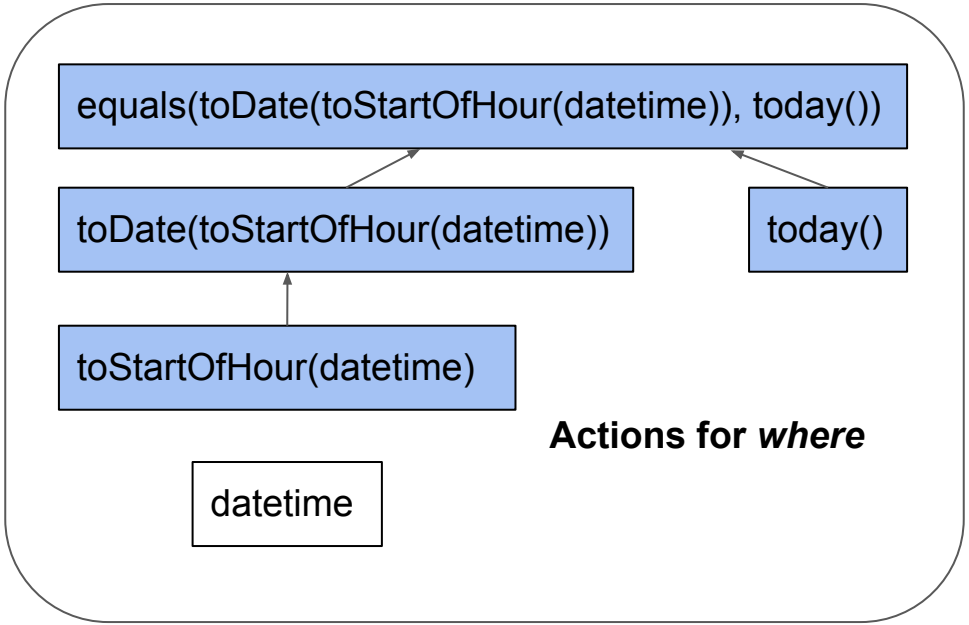


Query Analysis (Matching Aggregate Projection)

Projection *p_agg:*

```
Columns:

  toStartOfHour(datetime),
  domain,
  sum(bytes),
  avg(duration),


Primary Keys:
  toStartOfHour(datetime), domain
```

Projection *p_agg* still matches and becomes a candidate. **Complete Match!**



Query Analysis (Projection Selection)

- For every candidate, do primary key index analysis and cache the result.

Query Analysis (Projection Selection)

- For every candidate, do primary key index analysis and cache the result.
- Select the projection that reads the least amount of data.
    a. We don’t need to tell if it’s *normal* or *aggregate*. Less data is better.
    b. We will use cached result to avoid index reanalysis.

Query Analysis (Projection Selection)

- For every candidate, do primary key index analysis and cache the result.
- Select the projection that reads the least amount of data.
    a. We don’t need to tell if it’s *normal* or *aggregate*. Less data is better.
    b. We will use cached result to avoid index reanalysis.
- If a projection is selected, query pipeline will be extended to read both projection parts and ordinary parts.

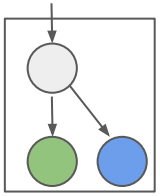
How Projection Achieves Consistency?

INSERT **SELECT** **MUTATION**

How Projection Achieves Consistency?

INSERT

When inserting one block of data, it's used as the source of all defined projections and is derived into a set of projection blocks. These blocks form the new data part together.



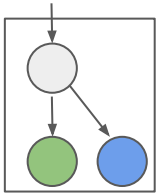
SELECT

MUTATION

How Projection Achieves Consistency?

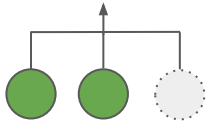
INSERT

When inserting one block of data, it's used as the source of all defined projections and is derived into a set of projection blocks. These blocks form the new data part together.



SELECT

When a projection is used, we build two different query pipelines to read from projection parts (materialized) and base parts (missing), and merge the result on the fly with negligible overheads.

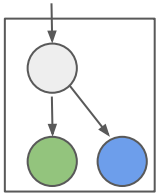


MUTATION

How Projection Achieves Consistency?

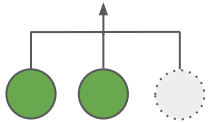
INSERT

When inserting one block of data, it's used as the source of all defined projections and is derived into a set of projection blocks. These blocks form the new data part together.



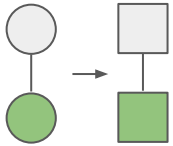
SELECT

When a projection is used, we build two different query pipelines to read from projection parts (materialized) and base parts (missing), and merge the result on the fly with negligible overheads.



MUTATION

Projections record their column dependencies during creation. When any dependent columns are changed, projection materialization kicks in and the mutated part will end up with newly built projections.



Pros & Cons

Experiments!

Projection Pros & Cons

Pros

1. Strong consistency over SELECT, INSERT, UPDATE, DELETE, etc.
2. Automatically select the best projection via query analysis without rewriting query.
3. Nifty user experience by using to-be-accelerated queries to create projection.

Projection Pros & Cons

Pros

1. Strong consistency over SELECT, INSERT, UPDATE, DELETE, etc.
2. Automatically select the best projection via query analysis without rewriting query.
3. Nifty user experience by using to-be-accelerated queries to create projection.

Cons

1. Cannot pre-aggregate across parts
2. Cannot have different rules of TTL or different storage policies than base part
3. Cannot support JOINs

Projection v.s. Materialized View & AggregatingMergeTree

Feature	Projection	Materialized View	AggregatingMergeTree
Data/Schema Consistency	Yes	No	Yes (nonintuitive)
Query Analysis	Yes	No	No
Data Reordering	Yes	Yes	No
Detail Data	Yes	Yes	No
Non-blocking Insert	Yes*	No	No
Complex Queries (Joins)	No	Yes	No

Experiments: Aggregate Projection

Dataset: 35 Billion rows per day
Aggregate Query: *group by toStartOfTenMinutes(datetime), domain*
Aggregation/Base Ratio: 0.004%

Aggregate Functions used in Query	Query Duration (1 thread)		Query Duration (24 threads)	
	Base	**Projection** *GROUP BY toStartOfTenMinutes*	**Base**	**Projection** GROUP BY *toStartOfTenMinutes*
countIf with filter	28.75s	0.03s	1.56s	0.02s
uniqHLL12	14.18s	0.05s	1.79s	0.05s
Three **simple aggregates**	50.29s	0.04s	3.43s	0.02s

Experiments: Aggregate Projection

Dataset: 35 Billion rows per day
Aggregate Query: *group by toStartOfTenMinutes(datetime), domain*
Aggregation/Base Ratio: 0.004%

Aggregate Functions used in Query	Query Duration (1 thread)		Query Duration (24 threads)	
	Base	**Projection** *GROUP BY toStartOfTenMinutes*	**Base**	**Projection** GROUP BY *toStartOfTenMinutes*
countIf with filter	28.75s	0.03s	1.56s	0.02s
uniqHLL12	14.18s	0.05s	1.79s	0.05s
Three **simple aggregates**	50.29s	0.04s	3.43s	0.02s

Experiments: Aggregate Projection

Dataset: 35 Billion rows per day
Aggregate Query: *group by toStartOfTenMinutes(datetime), domain*
Aggregation/Base Ratio: 0.004%

Aggregate Functions used in Query	Query Duration (1 thread)		Query Duration (24 threads)	
	Base	**Projection** *GROUP BY toStartOfTenMinutes*	**Base**	**Projection** GROUP BY *toStartOfTenMinutes*
⊕ ***countIf*** with filter	28.75s	0.03s ↑↑	1.56s	0.02s
⊗ ***uniqHLL12***	14.18s	0.05s ↑	1.79s	0.05s
Three **simple aggregates**	50.29s	0.04s ↑↑↑	3.43s	0.02s

Experiments: Aggregate Projection

Dataset: 35 Billion rows per day
Aggregate Query: *group by toStartOfTenMinutes(datetime), domain*
Aggregation/Base Ratio: 0.004%

Aggregate Functions used in Query	Query Duration (1 thread)		Query Duration (24 threads)	
	Base	**Projection** *GROUP BY toStartOfTenMinutes*	**Base**	**Projection** GROUP BY *toStartOfTenMinutes*
countIf with filter	28.75s	0.03s	1.56s	0.02s
uniqHLL12	14.18s	0.05s	1.79s	0.05s
Three **simple aggregates**	50.29s	0.04s	3.43s	0.02s

Experiments: Aggregation States

Parent part rows: 1118376

Projection_a rows: 9188

Aggregate Functions	Size
countIf(col_1 = 0)	16KB
count()	31KB
avg(col_1)	51KB
sum(col_1)	25KB
uniqHLL12(col_2)	18MB
uniqExact(col_2)	396MB

Projection_b rows: 13314

Aggregate Functions	Size
max(col_3)	35KB
quantileTDigest(0.9)(col_4)	3.9MB

Experiments: Aggregate Projection in Production

1. Collect frequent aggregation queries via normalized query analysis. Design and build multiple projections to pre-aggregate.
2. BI Dashboard with 12 sheets renders from 30 seconds down to **1 second**. Without projection only 4 sheets are rendered successfully.
3. Average additional storage consumption is less than **20%**.
4. Negligible insertion/merge impact. Still managed to insert **millions of rows per second**.

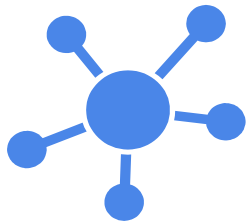
ClickHouse Projections are groups of columns which are defined by queries and used by queries. It's implemented by materializing queries into pieces of reusable data parts in a consistent way.

Future Works

1. Design and implement other types of projections
   a. As secondary index (store marks/offsets directly)
   b. As bitmap index (similar to Druid)

2. Expose projection as normal tables
   a. Support column encoding schemes
   b. Support skip indices

3. Store projection without base data
   a. Different TTL rules
   b. Different storage policies



Thanks!

Any questions?

You can find me at:

- amosbird in telegram
- amosbird@gmail.com

Feel brave?

You can try it out:

https://github.com/ClickHouse/ClickHouse/pull/20202

THANK YOU !

PERCONA
LIVEONLINE
MAY 12 - 13th
2021