ClickHouse Projections, ETL and more

郑天祺 博士, Amos Bird (Ph.D)，zhengtianqi@kuaishou.com

快手

About me

https://github.com/amosbird

- Active ClickHouse Contributor
  - ~300 valid PRs
  - ~40 Stack Overflow Answers
  - Doing some code reviews occasionally
  - Helping new ClickHouse developers
- Graduated from ICT CAS with a Ph.D degree in database
- Currently @kuaishou Data Platform Department

Outline

- Projections (MaterializedView in part-level)
- ClickHouse-ETL (Design new systems based on ClickHouse)
- Other Improvements
- Looking into the future

Projections

The Name of "Projection"

- Originated from Vertica (Don't confuse it with SQL Projection Op)
  - Projections are collections of table columns,
  - Projections store data in a format that optimizes query execution

- Our projection v.s Vertica
  - MergeTree* table == Vertica's super projection
  - Vertica only supports selected aggregate functions
    - SUM, MAX, MIN, COUNT, Top-K
  - We support arbitrary functions and their arbitrary combinations

Projection Definition

- Projection is defined by “SELECT … [GROUP BY] …”, with implicit “FROM <base_table>”

- Suppose we'd like to optimize the following query by pre-aggregation:

```
SELECT
    toStartOfMinute(datetime) AS _0, sum(i), avg(j),
    sum(i) / sum(j), topK(5)(id), quantile(0.99)(score)
FROM base_table
GROUP BY _0
```

- We can simply do

```
ALTER TABLE base_table ADD PROJECTION p
    (
        SELECT
            toStartOfMinute(datetime) AS _0, sum(i), avg(j),
            sum(i) / sum(j), topK(5)(id), quantile(0.99)(score)
        FROM base_table
        GROUP BY _0
    ) TYPE aggregate;
```

Projection DDL

- Newly added DDLs

```
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]) TYPE aggregate,
        ...
) ENGINE = [*]MergeTree ...

ALTER TABLE [db.]table ADD PROJECTION name (SELECT <COLUMN LIST EXPR> [GROUP BY]) TYPE aggregate;

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];
```

Projection Storage

- Projection is stored similar to skip indices
  - as a subdirectory inside part directory, named by projection name

```sql
CREATE TABLE base (`dt` DateTime, `col` int)
ENGINE = MergeTree PARTITION BY toDate(dt) ORDER BY dt;

ALTER TABLE base
    ADD PROJECTION prj_1
    (
        SELECT sum(col)
        GROUP BY toStartOfFiveMinute(dt)
    ) TYPE aggregate;

INSERT INTO base VALUES
    ('2020-10-24 00:00:00', 10),
    ('2020-10-24 00:00:00', 20),
    ('2020-10-24 00:00:00', 30);
```

```
data/data/default/base/
  ├── 20201024_1_1_0
  │   ├── checksums.txt
  │   ├── columns.txt
  │   ├── count.txt
  │   ├── col.bin
  │   ├── col.mrk2
  │   ├── dt.bin
  │   ├── dt.mrk2
  │   ├── minmax_dt.idx
  │   ├── partition.dat
  │   ├── primary.idx
  │   ├── prj_1
  │   │   ├── checksums.txt
  │   │   ├── columns.txt
  │   │   ├── count.txt
  │   │   ├── max%28col%29.bin
  │   │   ├── max%28col%29.mrk2
  │   │   ├── primary.idx
  │   │   ├── toStartOfFiveMinute%28dt%29.bin
  │   │   └── toStartOfFiveMinute%28dt%29.mrk2
```

Query Routing

- If a query can be deduced by a projection, it will be selected if:
  - allow_experimental_projection_optimization = 1
  - 50% of the selected parts contain materialized projection
  - total selected rows are less than base table

- If a projection is used, it will be listed in TRACE log similar to our index analysis
  - TODO we need better query explainer

Query Routing (Internals)

- Do projection query analysis up to WithMergeableState
- Ignore all aliases and normalize expression names (especially case-insensitive functions)
- Replacing expressions with columns which has the same name
- Check if projection provides all needed columns
- Rebuild query pipeline to read from projection parts
- TODO support prewhere, sample, distinct, group by with total, rollup or cube
- TODO support CNF normalization of projection predicates

Projection Merge

- Projection parts are merged exactly like normal parts
- If two parts don’t have the same projections, they cannot be merged
- In order to merge, we need explicit projection materialization or projection clear

Projection Materialization

- Building projections out from huge parts is expensive
  - Different sorting order
  - Aggregating over huge amount of data

- Implement as mutation so it will be run in background
- Implement as INSERT SELECT (projection name cannot start with “tmp_”)

```
tmp_mut_xxx/                          tmp_mut_xxx/
       |                                     |
       |--tmp_prj_a_1     ==>                |--prj_a
       |--tmp_prj_a_2
       |--tmp_prj_a_3
       |--tmp_prj_a_4
       |--tmp_prj_a_5
       |--tmp_prj_a_6
```

Materialization Optimization

- Avoid materializing unneeded columns
- Fast temporary parts removal (kudos to Alexey)
- Multi-run/multi-pass merge
  - Squash blocks to a minimum size threshold
  - Choose at most 10 parts to merge at a time
  - TODO loser tree to optimize merge sort

Projection v.s. Materialized View

Feature	Materialized View	Projection
Data Consistency	No	Yes
Schema Consistency	No	Yes
Query Routing	No	Yes
Query Index Optimization	No	Yes
Partial Materialization	No	Yes (but not recommended)
Complex Queries (Joins)	Yes	No (May support ARRAY JOIN)
Special Engines	Yes	No

Experiments

Projection: GROUP BY toStartOfTenMinutes(datetime)

Query	Duration (1 thread)	Duration (24 threads)
countIf with filters	28.75 sec	1.56 sec
countIf with filters (projection)	0.03 sec	0.02 sec
uniqHLL12	14.18 sec	1.79 sec
uniqHLL12 (projection)	0.05 sec	0.05 sec
3 aggregate functions	50.29 sec	3.43 sec
3 aggregate functions (projection)	0.04 sec	0.02 sec

Experiments (Space Consumption)

One part

	two dim projection	three dim projection	base
Rows	7223	20738	182779904

two dim projection	three dim projection	base
894M	1.1G	27G
293M	304M	4.7G
702M	781M	18G
923M	1.1G	26G
212M	238M	4.9G
110M	120M	2.1G
221M	260M	26G

Experiments (Aggregating States)

AggregateFunction	Size
countIf(col = 0)	16 kb
count()	31 kb
avg(col)	51 kb
sum(num)	25 kb
uniqHLL12(some_id)	18 mb
uniq(some_id)	396 mb

AggregateFunction	Size
max(cpu_idle)	369 kb
avg(cpu_idle)	450 kb
quantile(0.9)(cpu_idle)	20 mb
quantile(0.99)(cpu_idle)	20 mb

Projection in Production

- Dashboard rendering from 30 seconds to 2 seconds
- Average additional storage consumption is 20%
- Negligible insertion/merge impact

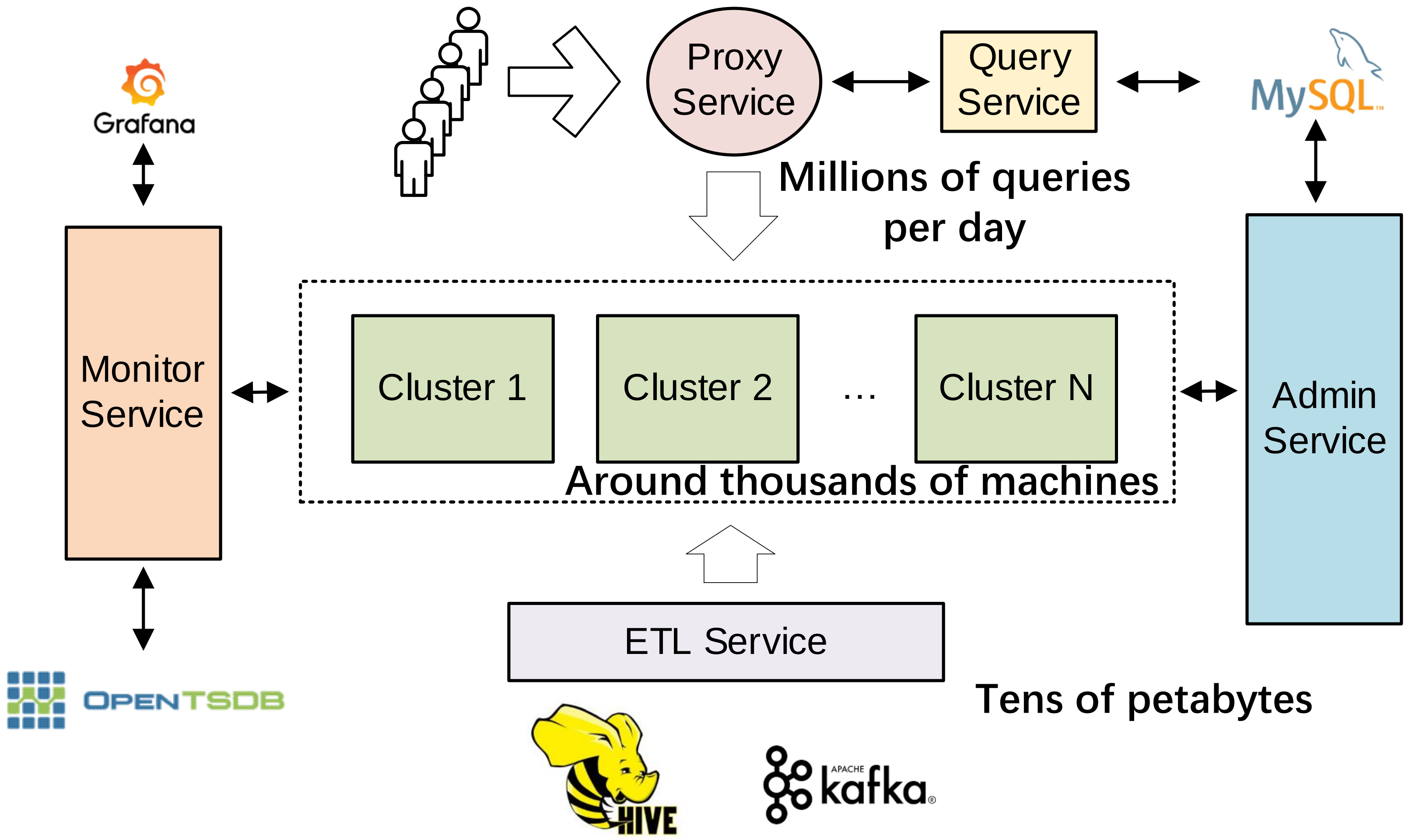
- Bonus Point
  - Use alias columns in Distributed table to match different projections with different granule of aggregation

Projection TBD

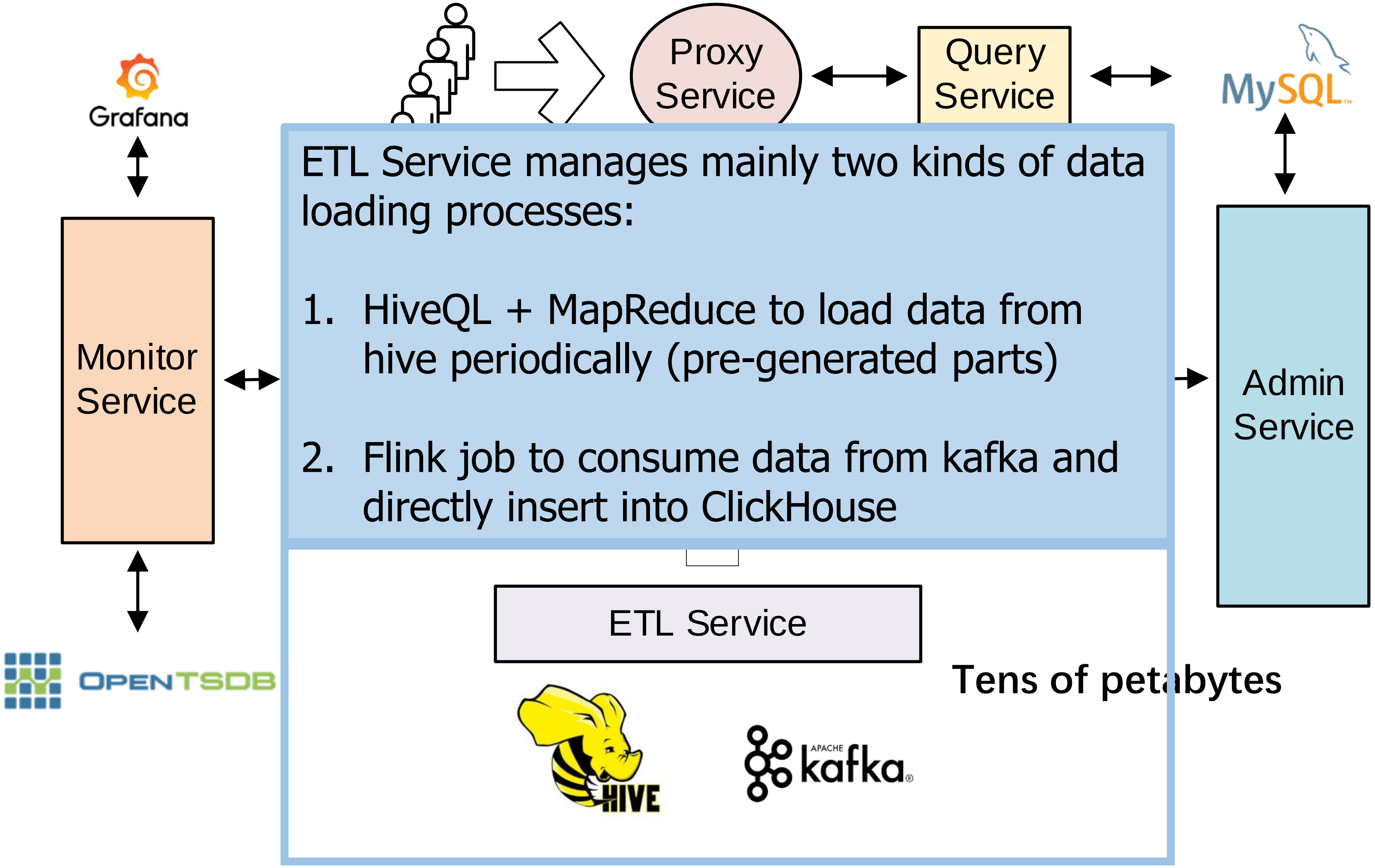
- Design and implement other types of projections
  - normal type: different ordering
  - secondary index: storing ScanRange directly
- ProjectionMergeTree
  - store projection without base table
- Support column encoding schemes
- Contribute

ClickHouse-ETL

Background: Our ClickHouse service



Background: Our ClickHouse service

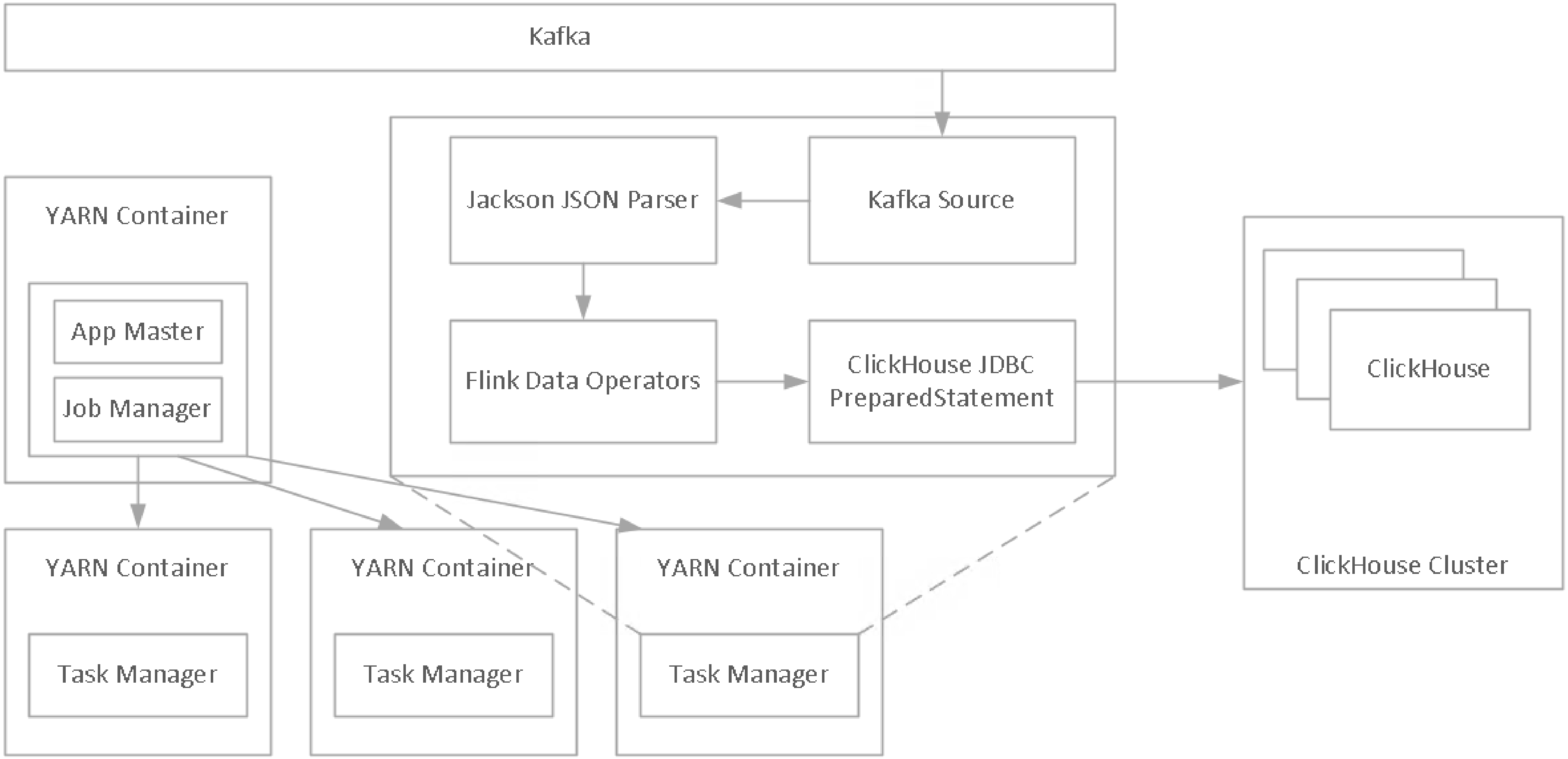


ETL Service manages mainly two kinds of data loading processes:

1. HiveQL + MapReduce to load data from hive periodically (pre-generated parts)

2. Flink job to consume data from kafka and directly insert into ClickHouse

Flink ETL Job



Main problems of Flink

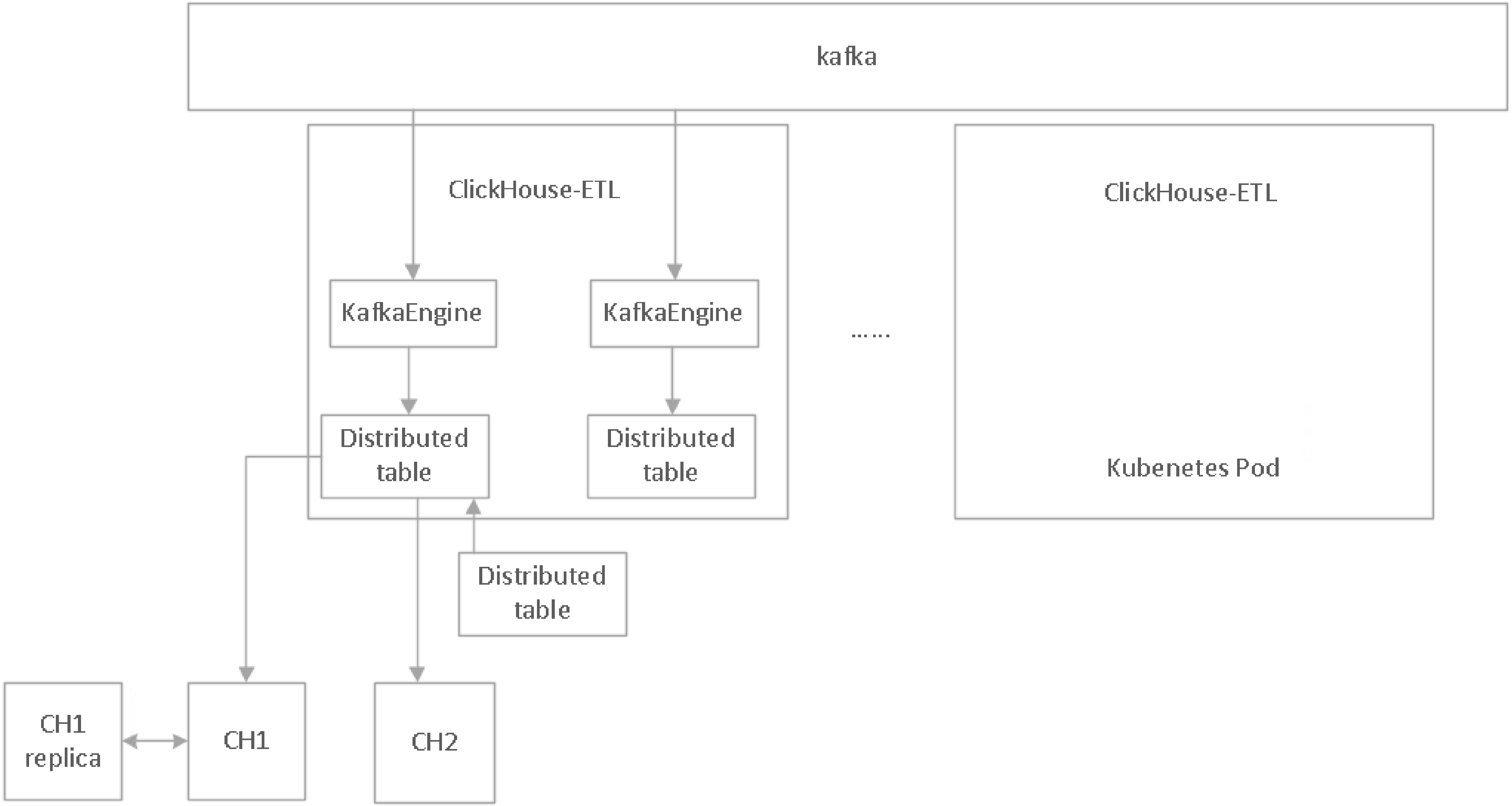
- Flink/Yarn scheduling is less flexible than k8s
- Java is slow in
  - Consuming from Kafka (kudos to librdkafka)
  - Parsing JSON (kudos SIMDJSON)
  - Inserting into ClickHouse
- Flink data transformation is cumbersome to use
- Java wastes memory (OOM when dealing huge messages)
- The pipeline lacks introspection capabilities

ClickHouse-ETL

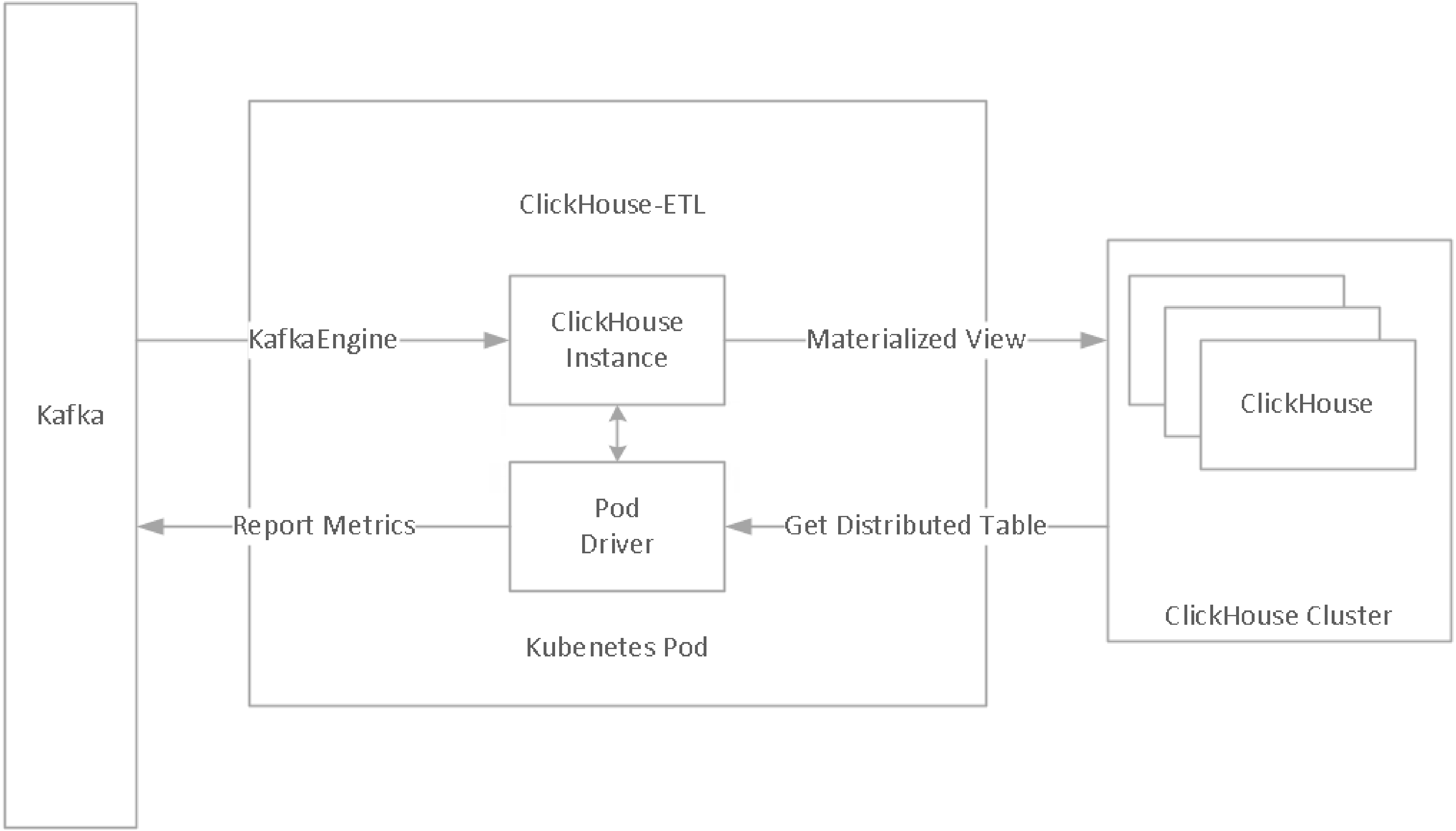
- Motivations (Our needs for real-time data ingestion)
  - Fast data input from Kafka
  - Ease of management
  - Reliable
  - Extensible

- An attempt of using ClickHouse as a tool to solve real problems

Introduce ClickHouse-ETL



ClickHouse-ETL Pod



Building Blocks

- Combine JSONExtract with untuple to inference schema on the fly
- Enhance StorageKafka and StorageDistributed to handle errors in better ways
- Utilize joinGet/StorageJoin with overwrite to update schema on the fly
- Take column transformers as the building blocks of ETL transformation grammar
- ETL pipeline is driven by MaterializedView (one per thread)

ETL Pipeline

```
-- Require one record per message
CREATE TABLE {database}.{table}_kafka_{idx} (line String) ENGINE Kafka SETTINGS kafka_format = 'BufferAsString', ...

CREATE MATERIALIZED VIEW {database}.{table}_mv_{idx} TO {database}.{table}
AS SELECT * {custom_transformation_str} FROM
    (SELECT * FROM
        (
            WITH JSONExtractWithErrors(line, joinGet('default.schema', 'value', 1)) AS t
            SELECT t.1 _json_parsing_errors, untuple(t.2, joinGet('default.schema', 'value', 2))
            FROM {database}.{table}_kafka_{idx}
        )
    {custom_filter_cond})
```

- `JSONExtractWithErrors`: attach two columns to record parsing errors
- `default.schema`: store the table schema and its JSON mappings
- `untuple`: unwrap Tuple result based on nested schema

Extended Table Schema

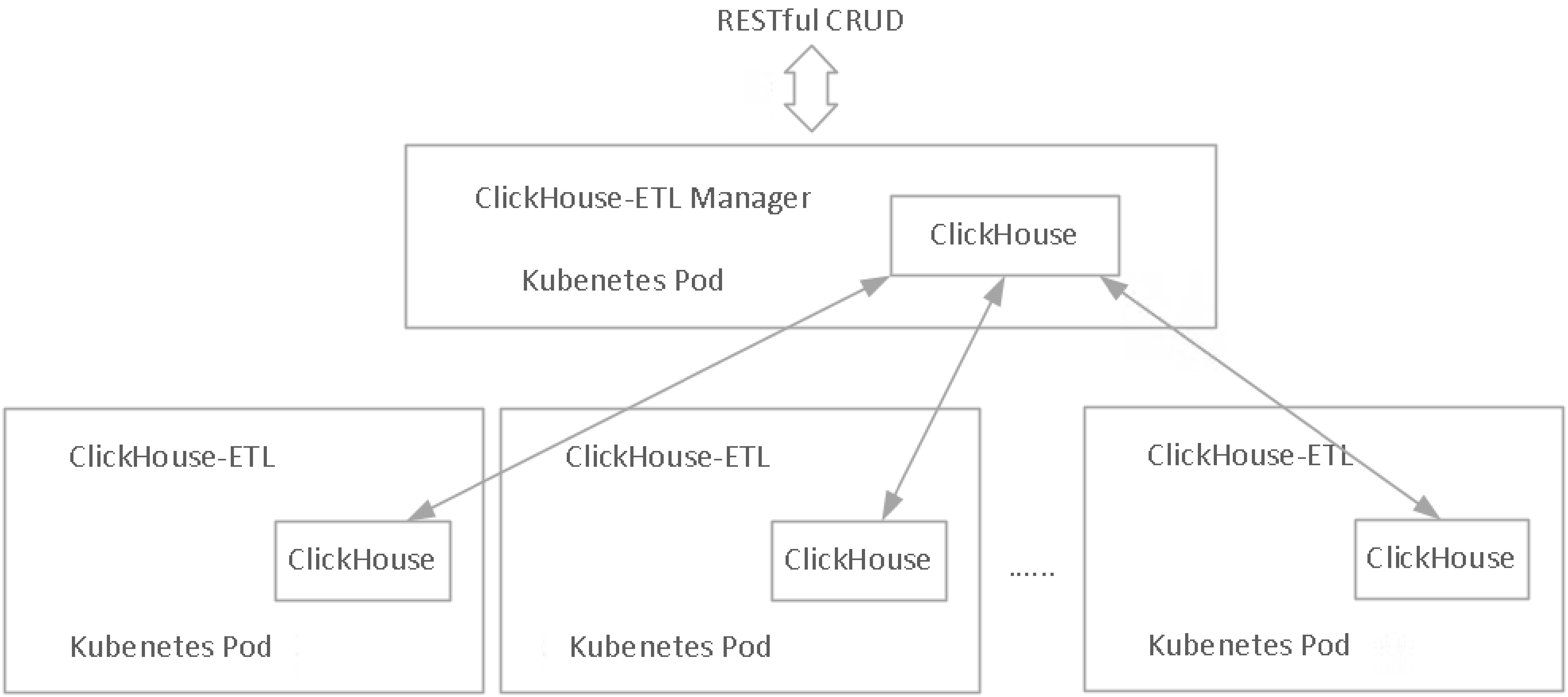
```
CREATE TABLE mytable
(
    id Nullable(UInt64), -- Nullable types will be inherited
    timestamp Int64, -- Millisecond timestamp
    datetime DateTime DEFAULT toDateTime(timestamp / 1000) COMMENT '$$', -- Ignore parsing datetime
    json_int Int32 COMMENT '$.json.int', -- Numeric types will be promoted.
    json_str String COMMENT '$.json.str',
    json_float Float64 COMMENT '$.json.float',
    json String COMMENT '$.`$json`', -- Store the json value as string
    int_val Int32 COMMENT '%.int_val', -- Parse int_val from string
    int_val2 Int32 COMMENT '%%' -- Parse int_val2 from string (JSON key is the same)
)
ENGINE = Distributed(mycluster, mydb, mytable_local, xxHash32(id));


SHOW SCHEMA mytable

Row 1:
──────
schema:  `id` Nullable(UInt64), `timestamp` Int64, `int_val2` String, `int_val` String,
         `$json` String, `json` Tuple(`float` Float64, `str` String, `int` Int64)

mapping: json.int AS json_int, json.str AS json_str, json.float AS json_float,
         `$json` AS json, int_val AS int_val
```

ClickHouse-ETL Manager



Introspection

- ClickHouse-ETL manager maintains a cluster "`clickhouse_etl`" of all Pods and keeps it update to date (1 minute)

```
SELECT * FROM cluster(clickhouse_etl, default.info) -- ETL Pod info
SELECT * FROM cluster(clickhouse_etl, system.etl_events) -- ETL metrics: drops, errors, etl
SELECT * FROM cluster(clickhouse_etl, system.kafka_info) -- Kafka metrics: lags, errors, etl
...
```

- Useful information is recorded in tables

Advantages

- ETL schema is recorded along with the table schema
- Writing to local Distributed table automatically honors the hashing key
- Schema changes are applied automatically
- Pods are stateless, easy to scale
- Fast, reliable, flexible, understandable
- Based on ClickHouse

Experiments

- Minimum cores to catch up with the data source

Rows per minute	Blocks per minute	Flink	ClickHouse-ETL
11.2M	739	800 cores	160 cores
30.2M	579	600 cores	100 cores
23.1M	50	60 cores	6 cores

Flink usually requires 1 core 4 GB mem
while ClickHouse-ETL uses 1 core 3 GB

ClickHouse-ETL TBD

- Exactly once semantic
- Partition adjustment
- Bulkloading with high availability

Other Improvements

Balance big parts among JBOD disks

- Define "Big Parts"
  - min_bytes_to_rebalance_partition_over_jbod
- Balance Big Parts in Partition Granule
  - Record current snapshot with emerging/submerging parts
  - Choose the best candidate(s) to land new big parts

Unbalanced partition '2021-01-28'

disk_name	sz
disk1	78.02 GiB
disk10	30.71 GiB
disk11	30.32 GiB
disk2	19.12 GiB
disk3	29.40 GiB
disk4	105.84 GiB
disk5	68.80 GiB
disk6	19.52 GiB
disk7	16.10 GiB
disk8	24.08 GiB
disk9	18.61 GiB

Balanced partition '2021-02-02'

disk_name	sz
disk1	32.45 GiB
disk10	42.52 GiB
disk11	31.12 GiB
disk2	38.54 GiB
disk3	41.92 GiB
disk4	30.83 GiB
disk5	38.83 GiB
disk6	43.63 GiB
disk7	43.59 GiB
disk8	38.85 GiB
disk9	43.94 GiB

Elasticsearch Storage Engine

- Based on ReadWriteBufferFromHTTP and SIMDJSON
- Push down primitive predicates
- esquery function
  - AST-level rewrite inside IStorage::read()
  - semantically equals to “not ignore”

Design of partition key/primary key

- Partition should be treated as the unit of data management

Partitioning is not to speed up selects - the main rule.
-- From Alexey Milovidov

- Primary keys should be ordered by usage rate

- Better to have keys with low cardinality come first in primary keys

Design of partition key/primary key

```
SELECT toStartOfMinute(datetime) _0,
       AVG(kbytes * 8 / duration) _1
FROM mytable
WHERE datetime >= '2021-01-06 06:00:00'
  AND datetime <= '2021-01-06 08:59:59'
  AND stream_id IN ('xxxxxxxx')
GROUP BY _0 settings max_threads = 1
```

Primary Key(s)	mark number	query (code)	query (hot)
datetime	256236 marks	152.877 sec	60.121
(datetime , stream_id)	21599 marks	34.249 sec	5.118
(toStartOfTenMinutes(datetime), stream_id)	125 marks	0.173 sec	0.042

Query Log Analysis

- Useful query information analysis
  - normalized_query, query_kind, databases, tables, columns

```
WITH quantiles(0, 0.5, 0.9, 0.99, 1)(query_duration_ms / 1000) AS t
SELECT
    min(query_start_time) AS s, max(query_start_time) AS e,
    normalizeQueryKeepNames(query) AS q, any(query) AS rq, count() AS cnt,
    countIf(exception != '') AS cnt_e, anyIf(exception, exception != '') AS e,
    t[1] AS t_min, t[2] AS t_p50, t[3] AS t_p90, t[4] AS t_p99, t[5] AS t_max
FROM
(
    SELECT
        query, exception, query_duration_ms, query_start_time
    FROM cluster(query_entrypoint, system.query_log)
    WHERE (query_kind = 'Select') AND (NOT has(databases, 'system')) AND is_initial_query
)
GROUP BY q ORDER BY cnt
```

- What if we don’t have the newer version ClickHouse?
  - Setup a local instance with all databases/tables, using engine Memory
  - Replay the queries (attaching event_time, duration as comment)

Clickhouse Client [Hidden] Features

- Open Editor (Alt-Shift-E)
- Bracket Pasting (-n without –m)
  - No need to provide semicolons
  - Better pasting experience
- Query Parameter with identifiers
- Customize Query ID Format

Miscellaneous features

- MergeTree-level settings
  - max_partitions_to_read
  - max_concurrent_queries/min_marks_to_honor_max_concurrent_queries

- Query Proxy Service
  - Global query quota and concurrency control

- Monitor On Cluster Hanging Issues
  - MaxDDLEntryID tracks the progress of on cluster DDLs
  - Check if any instance has fixed MaxDDLEntryID for a period of time while others don’t

Near future in kuaishou

- Extend and Explore Projections
  - Contribute to community
  - without fact table/as secondary indices/more storage scheme
- ClickHouse-ETL Exactly Once Semantic
- Subpartition
- Enhance Distributed Query Processing

Thank You!