Rebalancing data in shared-nothing distributed systems

Nicolae Vartolomei · 2022/01

This note describes the need for moving data between nodes of a distributed system, the challenges arising in the process, and a particular solution to the problem. The solution was contributed to the open-source ClickHouse DBMS in 2020–2021 as an experimental feature.

If you want to skip the intro, jump straight to the solution description.

Table of Contents

- Introduction
  - The need for (elastically) scalable systems
  - Scaling computing systems
  - Scaling distributed systems
  - Data rebalancing
  - Example: Apache HDFS approach to data rebalancing
  - Example: Ceph
- Our solution for rebalancing data in ClickHouse
  - ClickHouse overview
  - How to rebalance
  - Marking data to be moved with a unique identifier
  - Reading data and deduplication/conflict resolution
  - Distributed part pinning
  - Example
- Conclusion

Introduction

The need for (elastically) scalable systems

Some problems are well specified and have their inputs known from the beginning, e.g., find X and Y in this 100 PB dataset in N hours. Others are not well-defined and have requirements and input variables changing over time. Examples from the latter category include systems that accumulate business operational data, which tends to grow over time due to retention requirements, business growth, customer base growth, etc. Even for systems where predicting future requirements is an option, it often does not make economic sense to build that system today. Reasons include too much lump cash to invest; hardware tends to degrade; hardware gets cheaper over time; hardware gets faster.

The ideal approach is to build today the system that will satisfy the business requirements until the next procurement cycle, and as requirements change, add/replace the hardware to satisfy them.

Scaling computing systems

Consider systems that store and process data to answer questions about that data. As the quantity of data grows, the number of questions to answer (and their frequency) increases—the hardware resources requirements also grow. The system needs to be scaled up to meet the performance requirements.

There are four general strategies for scaling systems:

1. Redefine the problem to fit the software/hardware constraints.
2. Improve the software (faster algorithms, better data organization (compression, normalization), etc.).
3. Vertical scaling: replace old hardware with newer/faster hardware, add/upgrade CPUs/Memory/etc. to the machine.
4. Horizontal scaling: add more nodes to work on the problem.

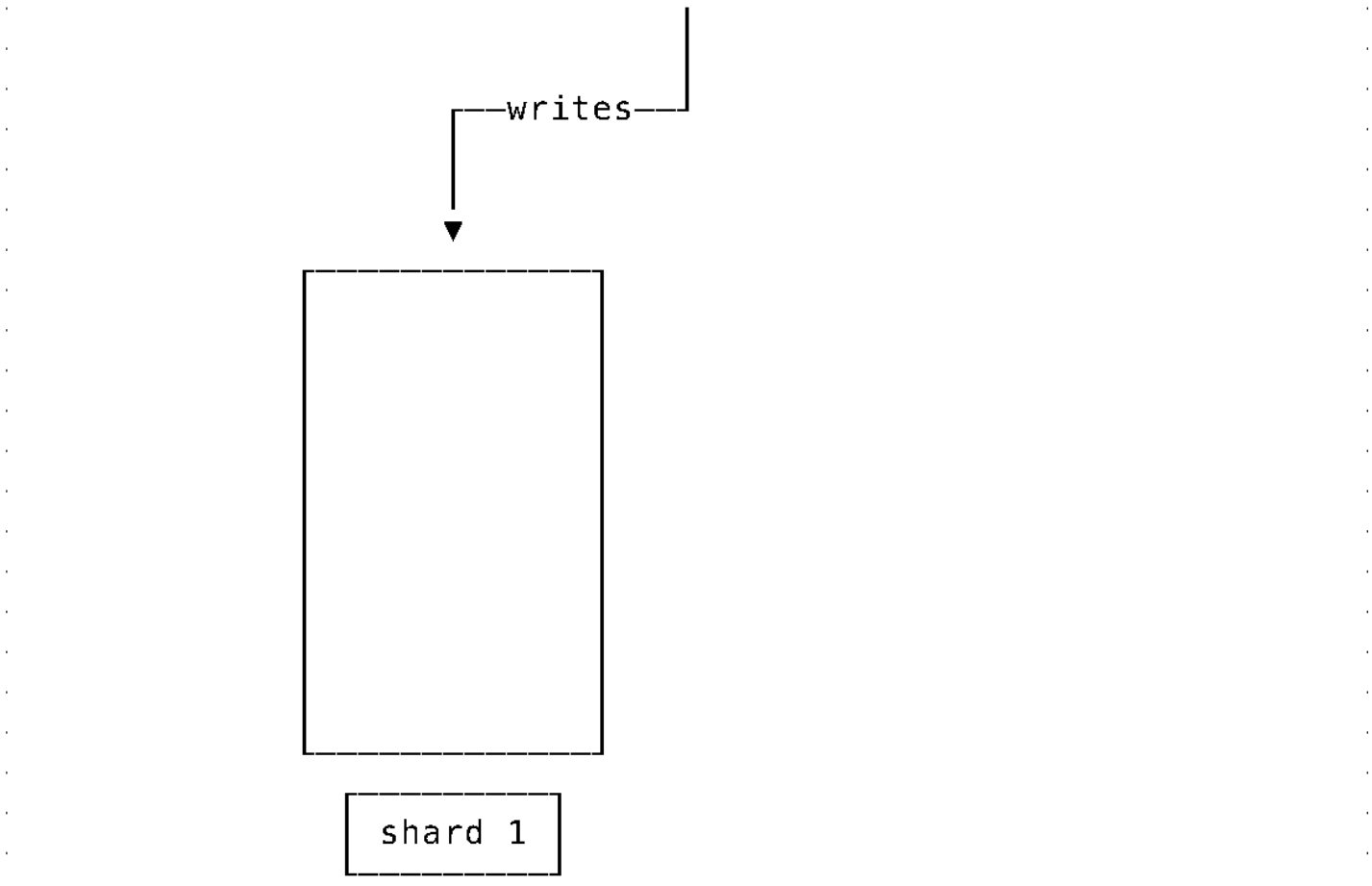
Scaling distributed systems

When a particular single node system is not enough to satisfy particular performance or reliability requirements, we turn our attention to distributed systems. Performance requirements examples: execution time, storage capacity.

Consider a system configuration which, among other things, is concerned with storing data. When the storage capacity is exhausted, a new node must be added.

Suppose newly arriving data is written to the resulting system uniformly. In that case, it will quickly reach the situation where one node is full and cannot accept more writes while the last added node is still underutilized.

Note: The case of random sharding is assumed here. Writes can go to any/many shards. Reads are aggregating data from all the shards (imagine that reads need to scan all data).



In some circumstances, such behavior is acceptable. And, can be dealt with using a simple strategy: write data only to newly added nodes. Such a strategy is far from ideal and becomes

untenable when additional requirements also have to be considered:

- Not only does the data quantity grow, but the rate of growth is also increasing. Now storage bandwidth and compute capacity need to be considered as well.
- The system manages multiple datasets, each having different growth rates and data retention requirements.

For the most flexibility, we need the ability to steer writes between shards and move data between shards (data rebalancing).

Data rebalancing

When data does not fit on an existing system anymore, we need to add more storage resources (servers) and move some data to the new servers.

The challenge is moving data transparently from the point of view of external observers (client).

In particular, at all times, the system must:

- **R1**: be responsive (no downtime)
- **R2**: not exclude data from results (no missing data)
- **R3**: not duplicate data in results

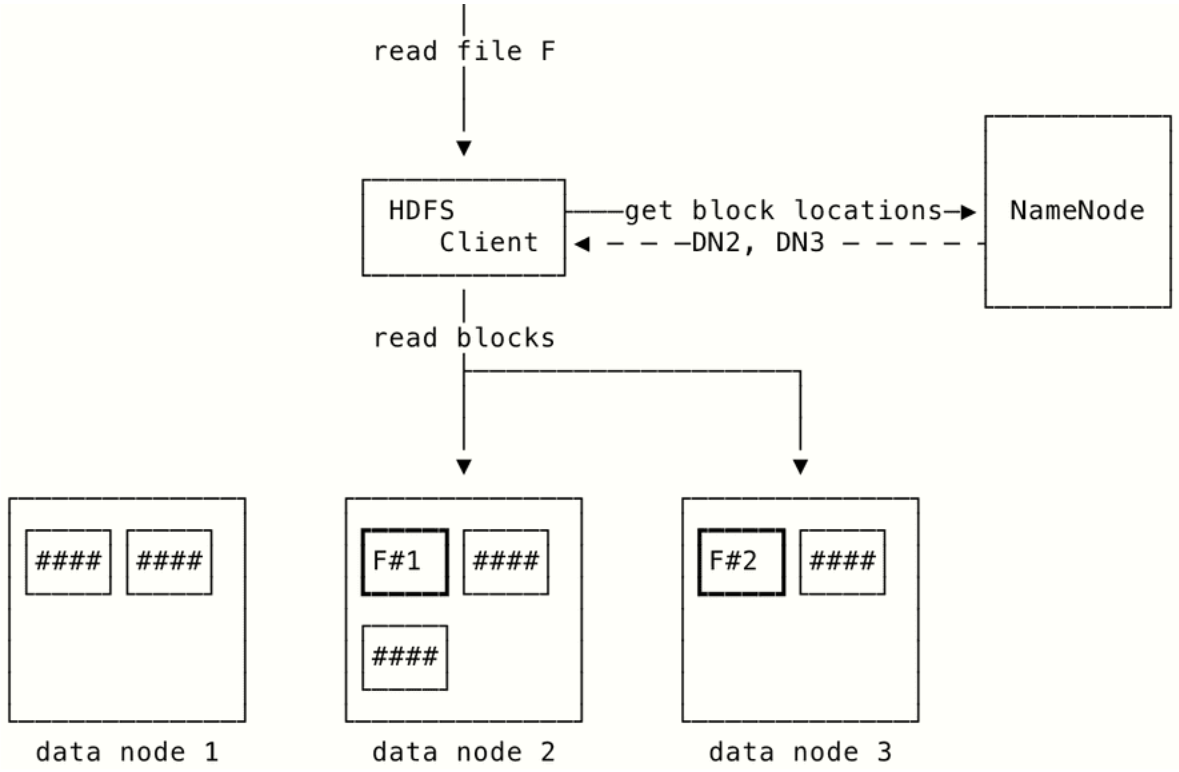
Let's go over possible naive approaches and see why they do not satisfy the requirements.

1. Shut down the system (like your bank does when daylight saving time kicks in), copy data from source to destination, remove data at source. **Violates R1**; shutting down the system is not acceptable.
2. Copy data from source to destination, remove data at source. **Violates R3**; both instances serve the same data after copy but before deletion.
3. Copy data from source to a temporary location, remove data at the source, copy data from the temporary location to destination. **Violates R2**; no instances serve data while it is stored only in the temporary location.

A form of coordination between client, source, and destination is required to make this work and satisfy the above-stated requirements.

Examples of how some systems try to solve this problem and how they add (or avoid) this co-ordination:

Example: Apache HDFS approach to data rebalancing



HDFS is a distributed file system for storing data organized as files and directories.

When writing or reading files, clients talk to a component called *NameNode* to identify *DataNode*(s) are responsible for a particular file (chunks of a file). It does that by maintaining a mapping between data identifiers and data nodes.

With the *NameNode* component available, the second (naive rebalancing) approach is extended with a single extra step to satisfy all data rebalancing requirements specified above.

HDFS solution: Copy data from source to destination. Update the location of data on the *NameNode*. Remove data from the source.

This approach comes with a downside. *NameNode* is in the request path. Adding an additional

request to the system results in latency overhead. And, more importantly, *NameNode* is a known bottleneck in large HDFS deployments.

Example: Ceph

Ceph is a distributed object, block, and file storage platform.

Clients request a *cluster map* from a centralized *Ceph Monitor* component. The *cluster map* is used to infer data location.

The approach is somewhat similar to the one seen in HDFS. The important difference is that the bottleneck seen in HDFS is avoided by using a lightweight *cluster map*, and a sharded metadata cluster.

If you want to learn more about it, see Ceph: A Scalable, High-Performance Distributed File System and CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data papers.

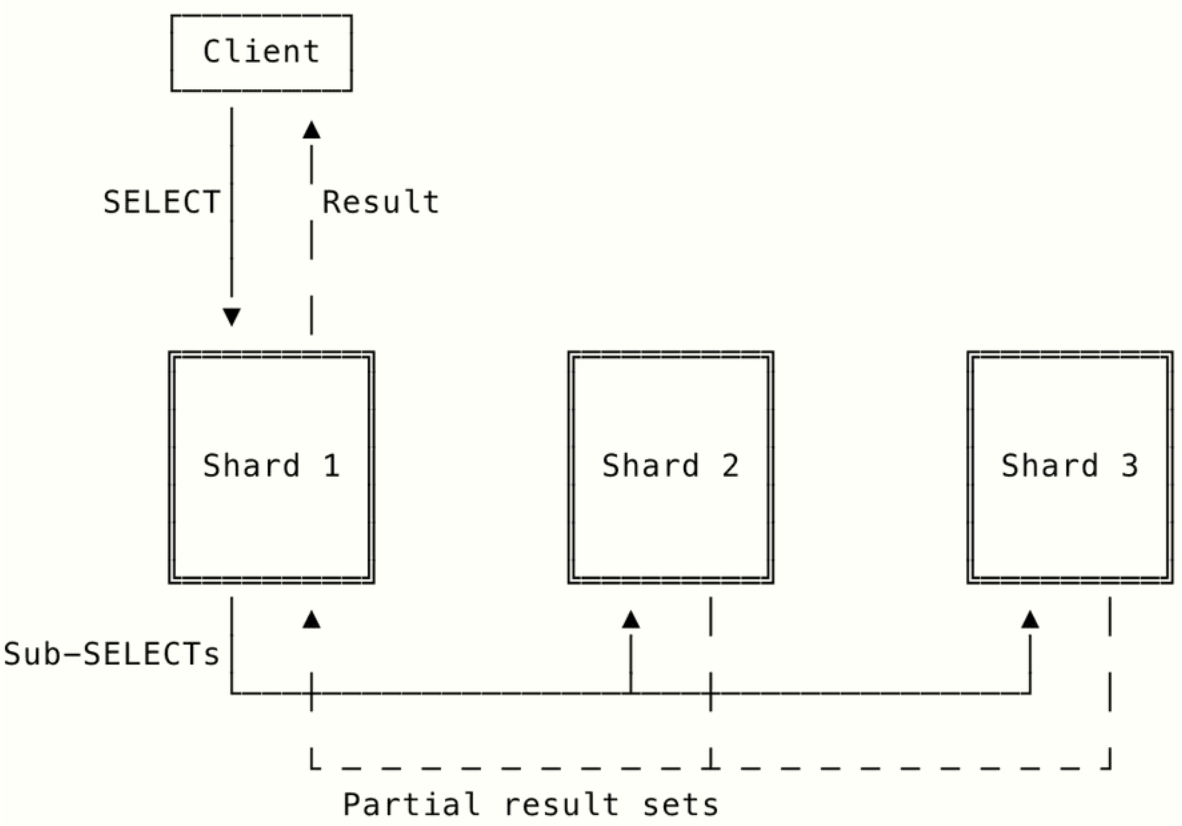
Our solution for rebalancing data in ClickHouse

ClickHouse overview

ClickHouse is an open-source, high-performance, distributed SQL OLAP database management system.

Usually, data is written with SQL's `INSERT` statements and read back with `SELECT`. It can ingest on the order of a million rows per second on good hardware and process on the order of a hundred millions rows per second for read queries.

These numbers scale near linearly in distributed mode. A single `SELECT` query can be split and processed by multiple nodes. Partial result sets are then aggregated together and sent to the client.



ClickHouse elasticity story lags behind. It is hard to scale a cluster (add more shards, remove shards) after it was created. The need for (elastically) scalable systems touches on the value of this feature.

We wanted to add the ability to move data between shards as one of the first steps in the elasticity story for on-premises ClickHouse deployments.

How to rebalance

I have spent more than a year searching for an elegant solution to this problem, and all paths seemed to lead to solutions similar to HDFS, or Ceph. Even if we leave aside that such a design introduces a possible bottleneck, a massive re-architecture of the ClickHouse would be required.

ClickHouse has a shared-nothing architecture; there is no coordination between different cluster shards. There is no centralized metadata store, as seen in HDFS or Ceph, which could be consulted before reading data. ClickHouse uses ZooKeeper for coordinating data replication within a shard, but that's it. Write path talks to ZooKeeper. Read path doesn't involve ZooKeeper. This is important for performance reasons.

Luckily, I wasn't the only one bothered by this problem. xjewer had a stroke of genius and came up with a proposal: Self-balancing architecture: Moving parts between shards #13574.

It seemed like magic because it ticks what looked like conflicting requirements:

- Proposes a solution for data movement which ticks all above-mentioned requirements:

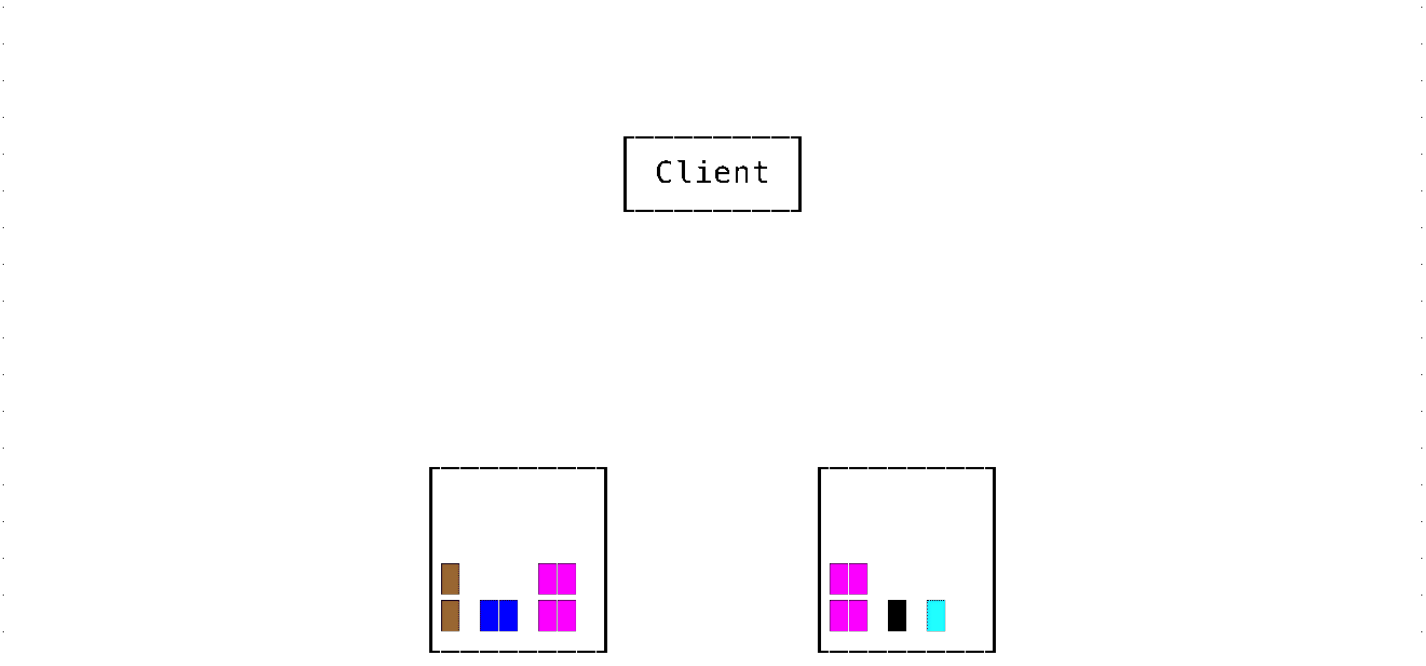
  > At all times, the system must:

    - **R1**: be responsive (no downtime)
    - **R2**: not exclude data from results (no missing data)
    - **R3**: not duplicate data in results

- Does not require introducing a centralized metadata store (HDFS), or advanced sharding schemes (Ceph).
- Does not require introducing coordination between shards (well, it does, but carefully and avoiding it in the hot path).

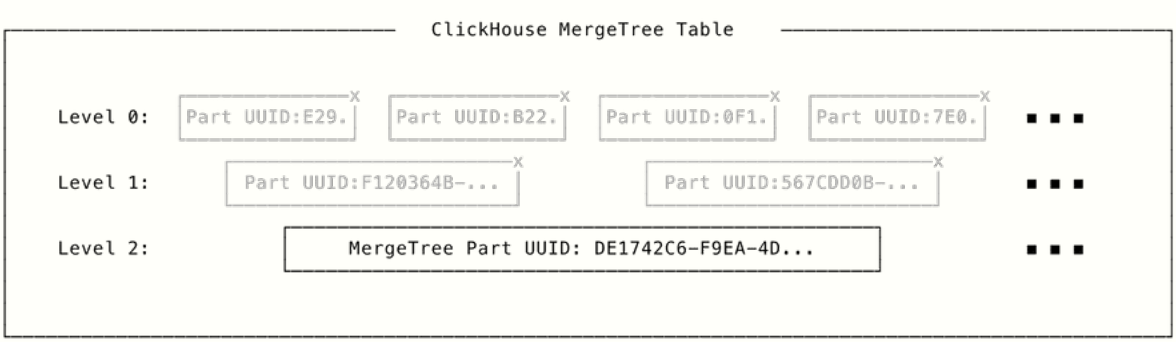
The meat of the proposal is as follows:

1. Mark data to be moved with a unique identifier
2. Copy data to the destination
3. Remove data from source
4. When reading data, use the unique identifier to discard duplicate data



Marking data to be moved with a unique identifier

ClickHouse MergeTree storage engine takes inspiration from the well known Log-structured merge-tree (LSM) family of data structures. Data is organized on disk into non-overlapping sorted runs (parts in CH vocabulary). A background process merges adjacent runs into larger runs to improve access performance, data compression, and in some cases, aggregate data.



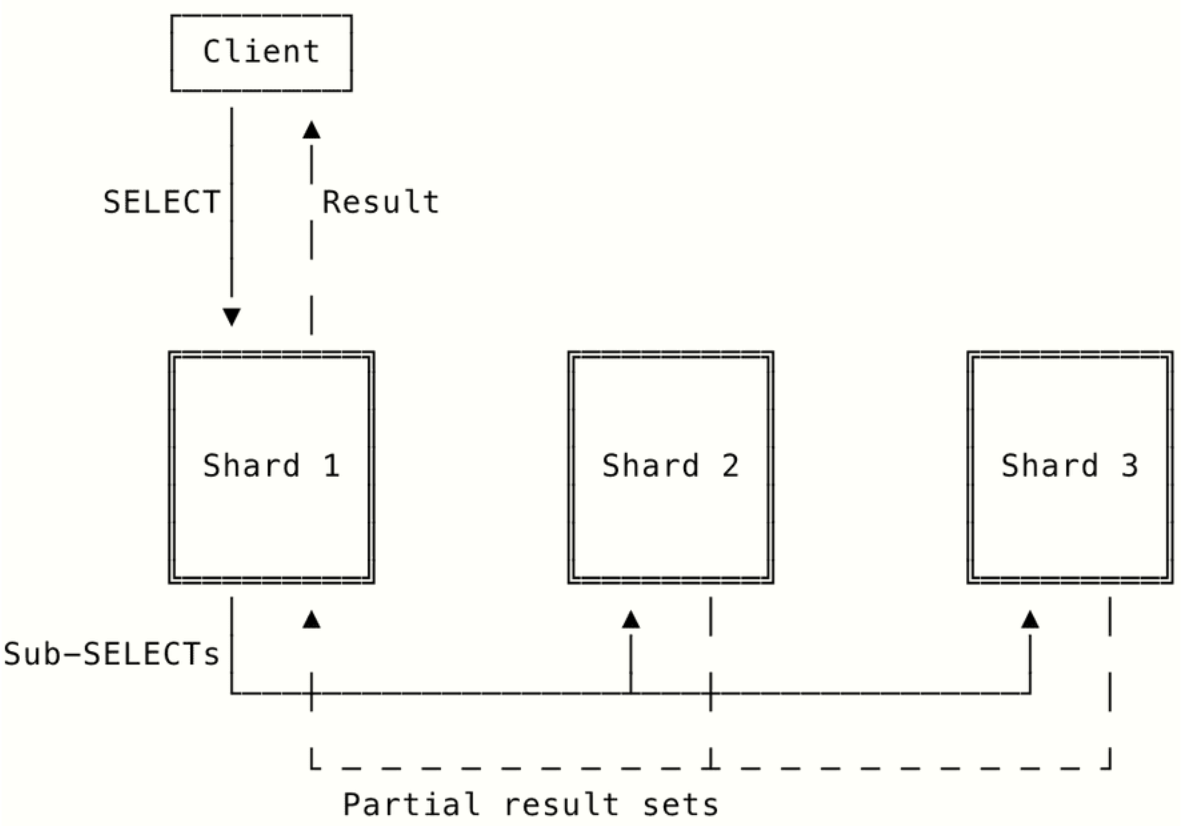
We decided to use the MergeTree Part as the unit of work for rebalancing and consequently for deduplication when reading.

Parts are usually large and contain a large number of rows (usually order of billions, up to 150 GB with the default configuration). Choosing part as the unit of work means that we don't have to split/rewrite large amounts of data on rebalance operations; we move the entire thing from one node to another. Also, we only have to assign/manage a single unique identifier for the purpose of deduplication to an entire batch of data (part). Another benefit is a lot of existing code operating at the part level, which we wanted to reuse.

We extended MergeTree Part structure with a new unique identifier which is generated uniquely each time a part is created (including parts created by merging existing parts or alters/mutations). Additional details are available in the Distributed part pinning section.

Reading data and deduplication/conflict resolution

Recall the diagram from the ClickHouse overview section:



When a ClickHouse node (root executor) receives a `SELECT` query it splits the query into sub-queries/leaf queries. Leaf queries are then sent to a replica of every shard in the cluster (leaf executor), where they are processed independently, and data is aggregated into intermediate states. The intermediate states are then further aggregated on the root executor into the final state, as was requested by the client.

Leaf executors record the unique identifiers of the processed parts and send these to the root executor. This is done just by inspecting in-memory metadata, before actually reading data from disks for query execution.

The root executor records the unique identifiers from the leaf executors and concurrently aggregates data received from them. When a conflicting unique identifier is received from a shard/leaf executor, the query for that executor is abandoned, and a new query is constructed with a condition to exclude the processing of the conflicting part data.

Exchanging unique part identifiers for each request is an unnecessary overhead. These are only needed for conflict resolution during part movement, and only for the parts currently moving between shards. As an optimization, we exchange only the identifiers of the parts currently moving/scheduled for move between shards.

Distributed part pinning

ClickHouse continuously looks for parts to be merged into larger parts. If we are moving parts between shards, we want to prevent that. Otherwise, it would be impossible to run conflict resolution on the root executor when processing results from leaves.

The set of pinned parts identified by their unique identifier is written to the coordination system. When ClickHouse tries to mutate a part, it asserts that its unique identifier is not present in the set of pinned parts.

In order to preserve the shared-nothing architecture on the read path, and avoid costly/slow calls to coordination system, we introduced new message types in the replication protocol and fencing in the part movement orchestration to guarantee that source and destination shards have their in-memory metadata up to date for the conflict resolution protocol to work correctly.

Example

The interface for moving data between shards looks like this:

```
ALTER TABLE stream_views
  MOVE PART '202201_42_96_8'
  TO SHARD '/clickhouse/shard_1/tables/stream_views';
```

The node on which the query is executed acts as source shard. Any replica of the source shard can be queried for part movement progress:

```
SELECT * FROM system.part_moves_between_shards;
```

```
┌─database─┬─table────────┬─task_name────────┬─task_uuid────────────────────────────┐
| default  | stream_views | task-0000000000 | 2b268e64-28c3-4f98-
ab26-8e6428c3bf98 | 2022-01-07 15:44:02 | 202201_42_96_8 | 5fac6788-
ad31-4016-9fac-6788ad316016 | /clickhouse/shard_1/tables/stream_views |
2022-01-07 15:49:05 | DONE  |         0 |                 |
└──────────┴──────────────┴──────────────────┴──────────────────────────────────────┘
```

For a complete example see the bundled test and the required configuration to enable this feature.

Conclusion

With this work completed, ClickHouse is more elastic for certain use-cases (random sharding, infrequent rebalancing). Moreover, it was done with what I think is the minimal number of changes to the system. It means as few potential controversial decisions points as possible, important for having a contribution accepted to an open-source project in a timely manner.

Use-cases where this approach falls short (e.g. when sharding by a key is used rather than random sharding, frequent rebalancing) I expect (speculate) to be addressed after ClickHouse will move to a cloud-native architecture. In my opinion this will require additional changes to the current design that will make the implementation much simpler compared to what is required today.

Thanks to Aleksei “xjewer” Semiglazov for coming up with the idea described in this note, reviewing my ClickHouse PRs and drafts for this note.

- ClickHouse Issue#13574: Self-balancing architecture: Moving parts between shards
- ClickHouse PR#16033: Add unique identifiers IMergeTreeDataPart structure
- ClickHouse PR#17348: query deduplication based on parts’ UUID
- ClickHouse PR#17871: Part movement between shards
- ClickHouse PR#29043: Part movements between shards improvements and cancel support

Nicolae Vartolomei