Fast, Scalable and Reliable Logging at Uber with Clickhouse

February 11, 2020

Uber

Agenda

- Mission and Goals of Logging
- Background and Challenges
- ClickHouse Evaluation
- ClickHouse Based Logging Architecture
- Questions

Mission and Goals of Logging

Vision of Reliability Platform

Uber engineers have the platforms, tools, and support to rapidly develop and confidently operate their services reliably at scale

Uber

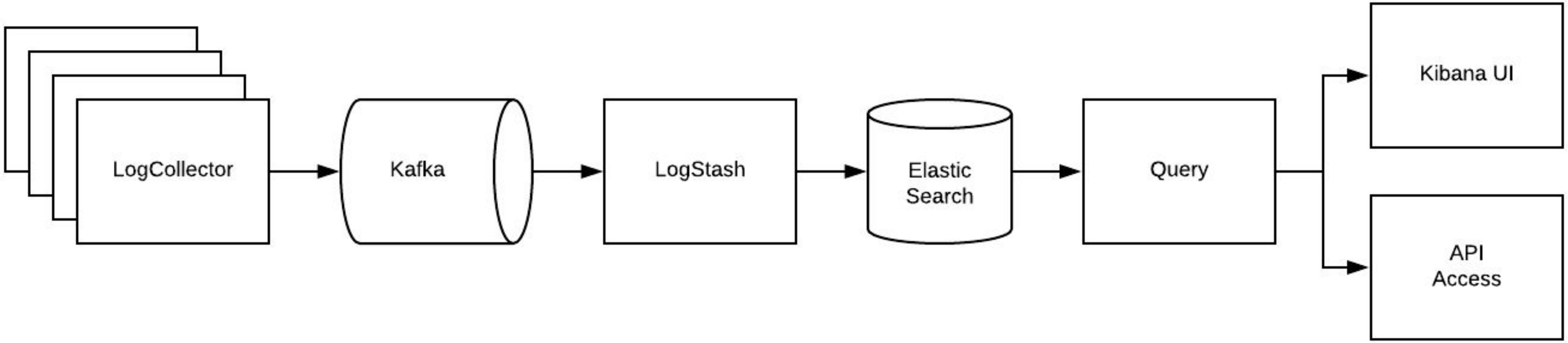
Logging Mission

- *Make it work*: Maximize the speed at which engineers can act upon operational data
- *Make it scale*: Scale to meet today's needs and tomorrow's growth
- *Make it cheap*: Ensure a consistent and sustainable cost model

Uber

Background and Challenges

High-Level Architecture



Current Scale of Logging

- We collect a lot of things
  - **Thousands of** Services emitting **hundreds TB** logs per day
- We store a lot of things
  - **Low Petabytes** of logs stored
- We query them for real-time debugging, offline troubleshooting, analytics, etc.
  - **hundreds queries / s** from dashboards and API queries

Uber

Challenge: Developer Productivity

- Logging users want schema-free logging
  - Services can write logs with very different structures
  - Log schema evolves over time (new fields, changing field types, etc.)
- ElasticSearch requires a consistent schema per index
- Type conflicts: log field type inconsistency => ElasticSearch exceptions
  - Disable field, drop logs
  - Can significantly degrade ES performance and affect co-tenants in cluster
  - Requires back-and-forth between logging team and service owner to fix

Uber

Challenge: Performance

- Performance challenges
    - End-to-end ingestion latency
        - >2 minute latency for large indices
        - ES indexes data in batches, reducing batch time can result in significant performance degradation due to higher indexing overhead
    - Query latency
        - Poor resource isolation, expensive queries can significantly degrade cluster performance, and sometimes render cluster unresponsive even after query stops.

Uber

Challenge: Scalability and Operability

- High cost makes it expensive to scale
- Operational challenges at scale
  - Running multiple ES clusters
    - Having too many nodes in one cluster puts strain on the master node
  - General reliability issues
    - JVM heap lockup after a single expensive query requires bouncing the entire cluster

Uber

ClickHouse Evaluation

Evaluation Setup

- Ingested production logs from Kafka into candidate storage cluster under evaluation
- Continuously evaluating common types of production queries against candidate storage cluster:
  - Group by query: "For time range X and services Y, give me the top 5 most frequently accessed endpoints matching filter Z"
  - Histogram query: "For time range X and services Y, give me the number of log events per minute matching filter Z"
  - Raw query: "For time range X and services Y, give me the most recent 500 logs matching filter Z ordered by time"

Uber

Key Observations about Logging Use Cases

- Observations:
  - Schema-free logging is highly desirable
  - Number of logs queried << number of logs ingested
  - Number of log fields accessed << number of log fields stored
  - Indexing on all fields incurs significant performance overhead
- A columnar storage that
  - Provides mechanisms to support schema-free logging for developer productivity
  - Indexes on only the necessary fields but no more
    - Performance for querying indexed fields
    - Efficiency for not indexing all fields

What's ClickHouse?

- An open-source, distributed, high performance columnar DBMS
- High throughput ingestion with asynchronous segment merging, requires no locks during concurrent writes
- High performance parallelized query execution
- Supports a query language covering majority of SQL capabilities (GROUP BY, ORDER BY, JOIN, etc.)
- Built-in clustering mechanism supports configurable sharding, multi-master shard-level writing and replication, and distributed query processing

"Why Clickhouse: Ingestion"

- Writes 3x - 4x throughput compared to ES
- Ingest performance scales close to linearly to cluster size
    - Writes evenly distributed across the cluster results in even load distribution
    - Independent shard design maximizes single-node performance as cluster size increases
    - Multi-master replication ensures no SPOF in design

Uber

"Why Clickhouse: Query"

- Data scanning speed during query processing
  - ~5x query speed of ES
  - Vectorized execution and parallelized processing across cores achieves high scanning speed
- Expected to support high hundreds in QPS
- Better control on resource allocation
- Increases in query concurrency beyond max levels does not cause cluster instability
  - OTOH, ES may experience cluster-wide lockup due to high query load even after query is cancelled / timeout

Uber

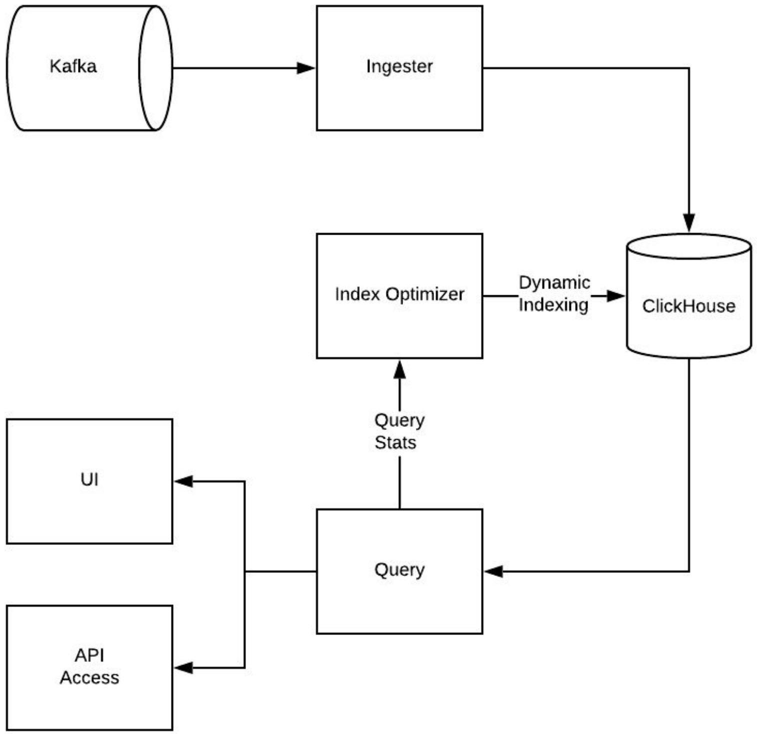
"Why Clickhouse: Storage"

- Configurable column-level compression algorithm
  - LZ4, ZSTD, …
  - Allows more efficient storage, faster disk I/O, and bigger raw dataset to fit in filesystem cache
- Compression ratios
  - LZ4: 3x for logs with complex schema, 20x for small, structured logs
  - ZSTD: 2x - 3x better compression ratio than LZ4 at 15% higher CPU cost
- Data are partitioned by configurable partition keys allowing pruning large amount of data partitions during query execution.
- Supports dynamically building and asynchronously backfilling materialized columns and data skipping indices, further speeding up log field queries

Uber

ClickHouse Based Logging Architecture

High-Level System Architecture



Ingestion

- Consumes log events from Kafka, and flatten JSON logs into structured fields.
  - Honor field types: foo.String vs foo.Number
- Buffers log events into big batches, and routes them to the proper ClickHouse tables.
- No need to sanitize logs to prevent type conflicts

Uber

Dynamic Indexing

- By default, ingest everything, index nothing.
  - Basic query performance with base table schema with native ClickHouse functions
  - < 5% of log fields are ever accessed, don't pay the price for indexing the other 95%
  - No blind indexing == High ingestion throughput
- Indexing is still important and necessary for the 5% to ensure low query latency.
  - Much less data scanned at query time
  - Taking full advantage of columnar storage and vectorized processing.
- Dynamic indexing
  - Adaptive to query patterns: Index log fields that are frequently queried.

Materialized Columns

- Materialized columns derive their values from base columns
- Can be created or dropped at runtime
    - `ALTER TABLE <table_name> ADD COLUMN "endpoint.String" ...`
- When a materialized column is created
    - Automatically populated for new incoming rows
    - Asynchronously backfill from historical values during data merging
    - Querying such column will automatically "do the right thing"
- Scanning speed for materialized columns
    - >10x faster than scanning base schema

Uber

Data Skipping Indices

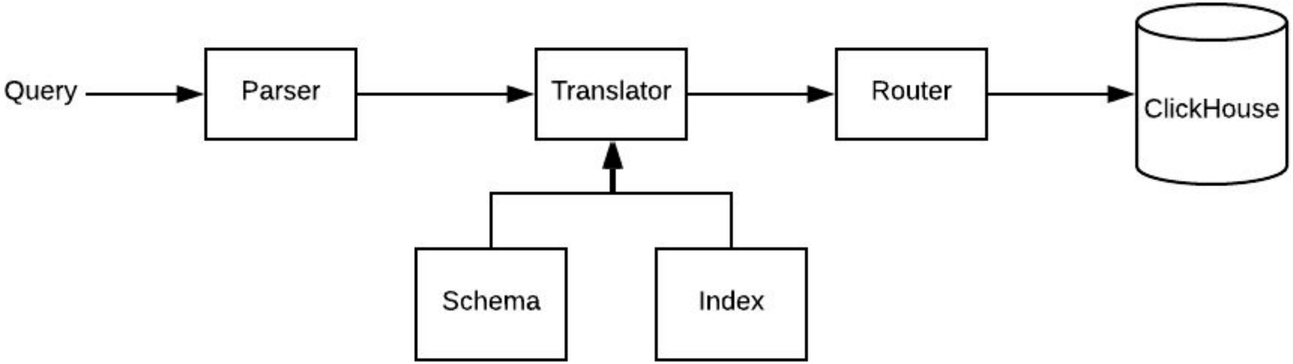
- Types of data skipping indices
  - Token-based and n-gram based bloom filter indices: `equals`, `in`, ...
  - MinMax indices
  - Set-based indices
- Using the right indices can significantly speed up queries
  - Token-based bloom filter index for UUID matches
  - 15x query latency reduction compared to when no index is used

On-Demand Indexing

- Adaptive to query pattern and user input on the fly
  - Feedback loop in minutes
  - New incoming data are immediately indexed
  - Asynchronously backfilling indices for historical data during segment merging, can be accelerated if needed

Query

- Parses incoming query and translates it into a SQL expression understood by ClickHouse
  - Uses schema to determine available fields and their types
    - Conflict resolution when a field has multiple types
  - Favors materialized columns, fall back to base schema scans if unavailable
  - ClickHouse makes use of data skipping indices transparently if available



Uber

Query (Cont'd)

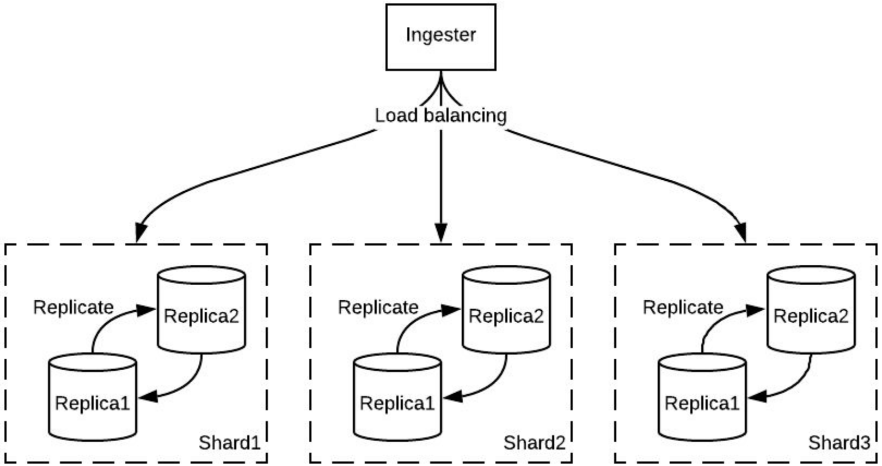
- Configurable query execution
  - Resource allocation per query
  - Workload isolation
  - Cost accounting
- Linearly scalable with more resources
  - Able to provide better performance for high priority queries by allocating more resources
- Fine-grained control for distributed query processing
  - Skip shards with errors
  - Timeout slow shards early
  - Strategy to pick from replicas in a shard

Clustering

- Fundamental clustering functions out of the box
- Uniform shard distribution, rack-aware shard topology
- Writes evenly distributed across nodes ensuring balanced ingestion load cluster wide
- "Distributed table" primitive enables distributed queries across shards and merging results happen transparently
- Efficient, multi-master replication ensuring little to no write throughput degradation with replication enabled

Clustering: Ingestion

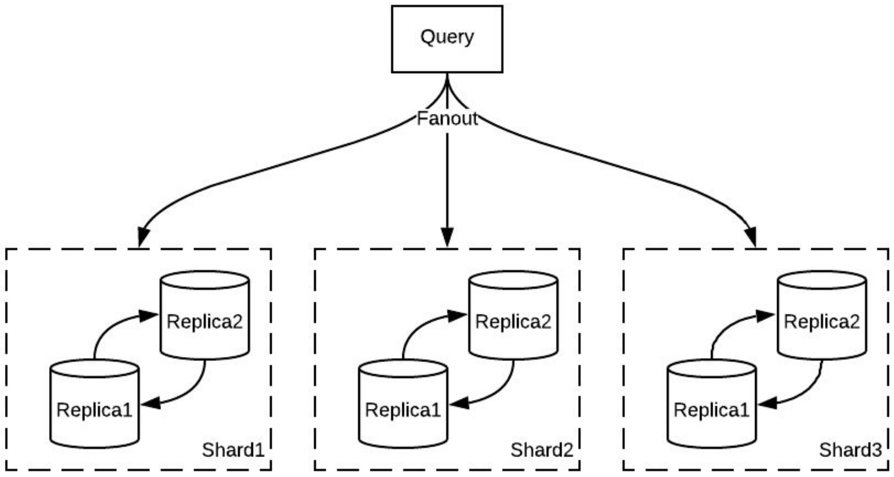
- Writes evenly routed to any node in the cluster
- Data replicated asynchronously to the peer in the same shard



Uber

Clustering: Query

- Distributed query can be issued to query nodes
- The node fanouts sub-queries to all shards in the cluster
- The node aggregates the results from the sub-queries and return



Uber

Unified Multi-Tenant Storage Platform

- ClickHouse natively supports zero lock contention among concurrent reads and writes
- Service placement: single-tenant vs multi-tenant
  - Isolate heavy log producers, heavy log consumers
  - Co-locate everything else
  - Limit the impact of co-location, add service in order-by
- Workload isolation
  - Configure query parallelism per query
  - Eventually limit total query resource usage per node
  - Query cost accounting, defense against expensive queries

Uber

Q & A