# Building Real-Time Data Lake with ClickHouse and Altinity Antalya

Alexander Zaitsev – Altinity, Co-Founder and CTO


X


AUG 28 YEREVAN


Run Open Source ClickHouse® Better

# Altinity.Cloud  Enterprise Support

# Three Types of Innovation

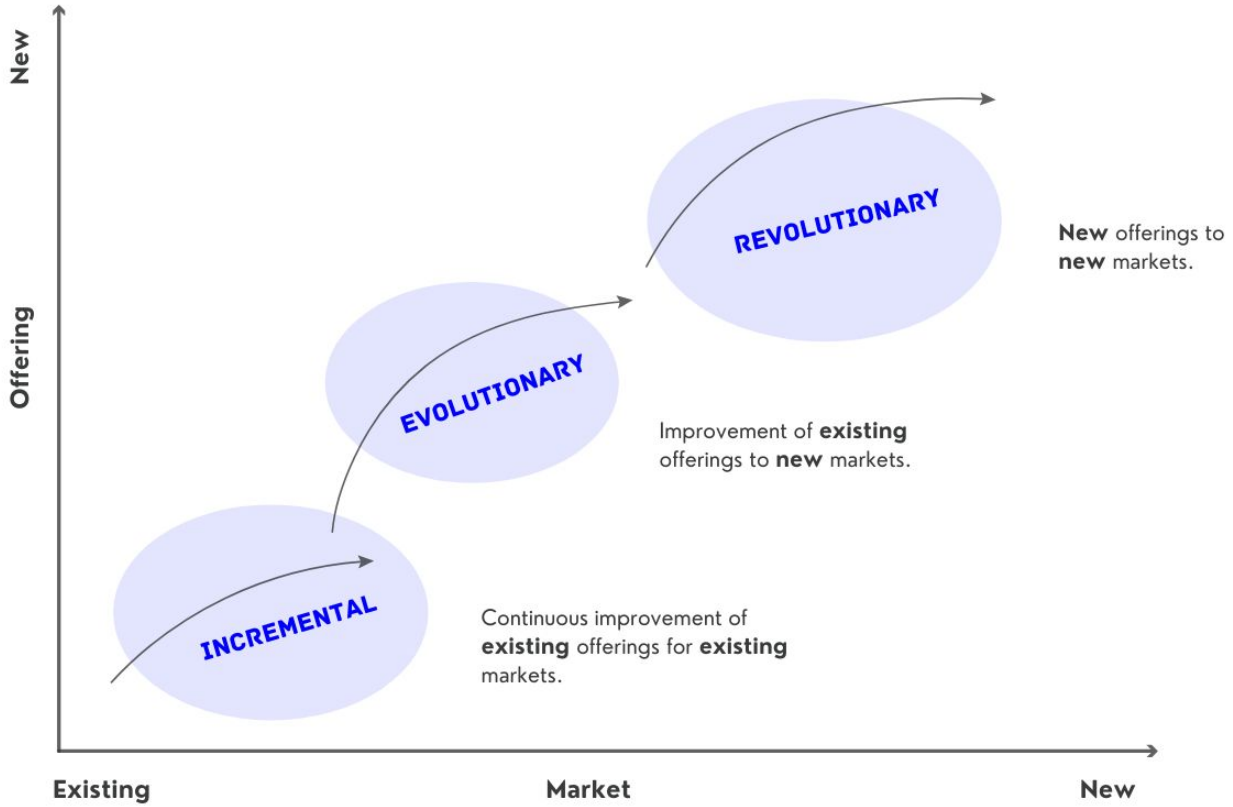



Image from https://intranel.com/resources/innovation-for-established-businesses

# Listen to What Users Say

# ClickHouse is a famous real-time analytic database

Statically linked C++ binary

Run anywhere

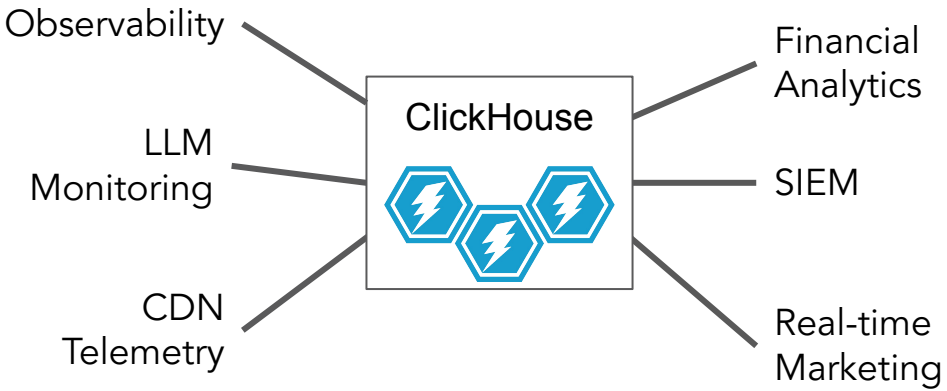
Column storage

Parallel and vectorized execution

Scales to many petabytes

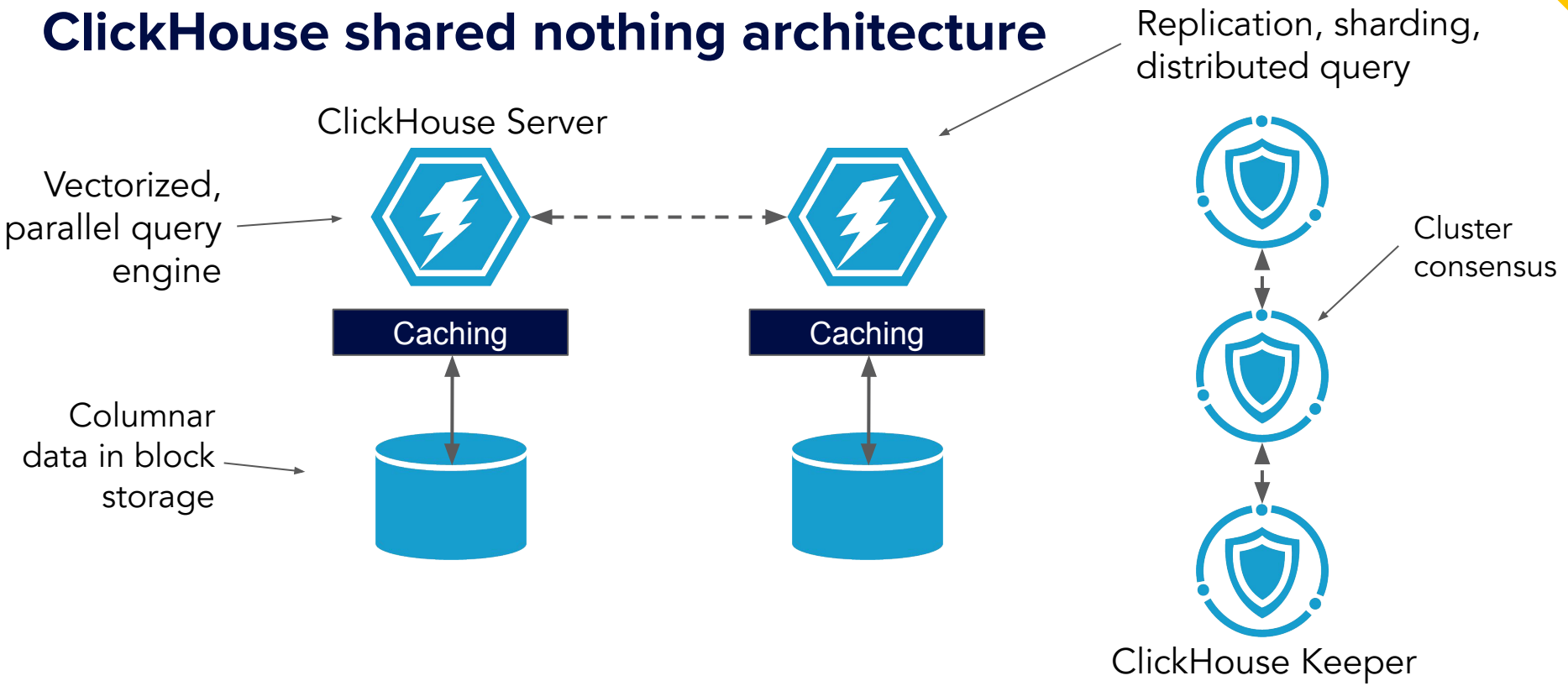
Apache 2.0  license

41.8k GitHub stars (≈ Spark)



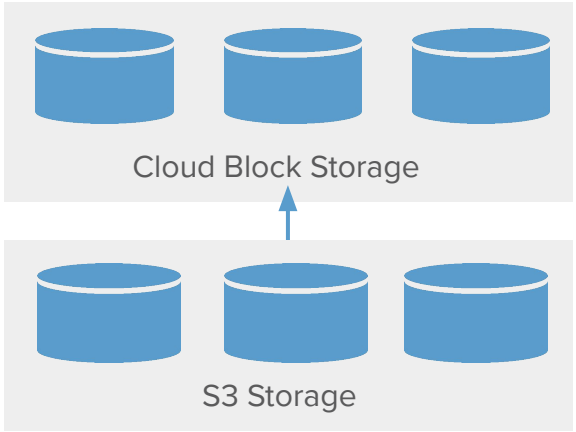

**Stable, Sub-Second Response**

# ClickHouse shared nothing architecture



# The Problem

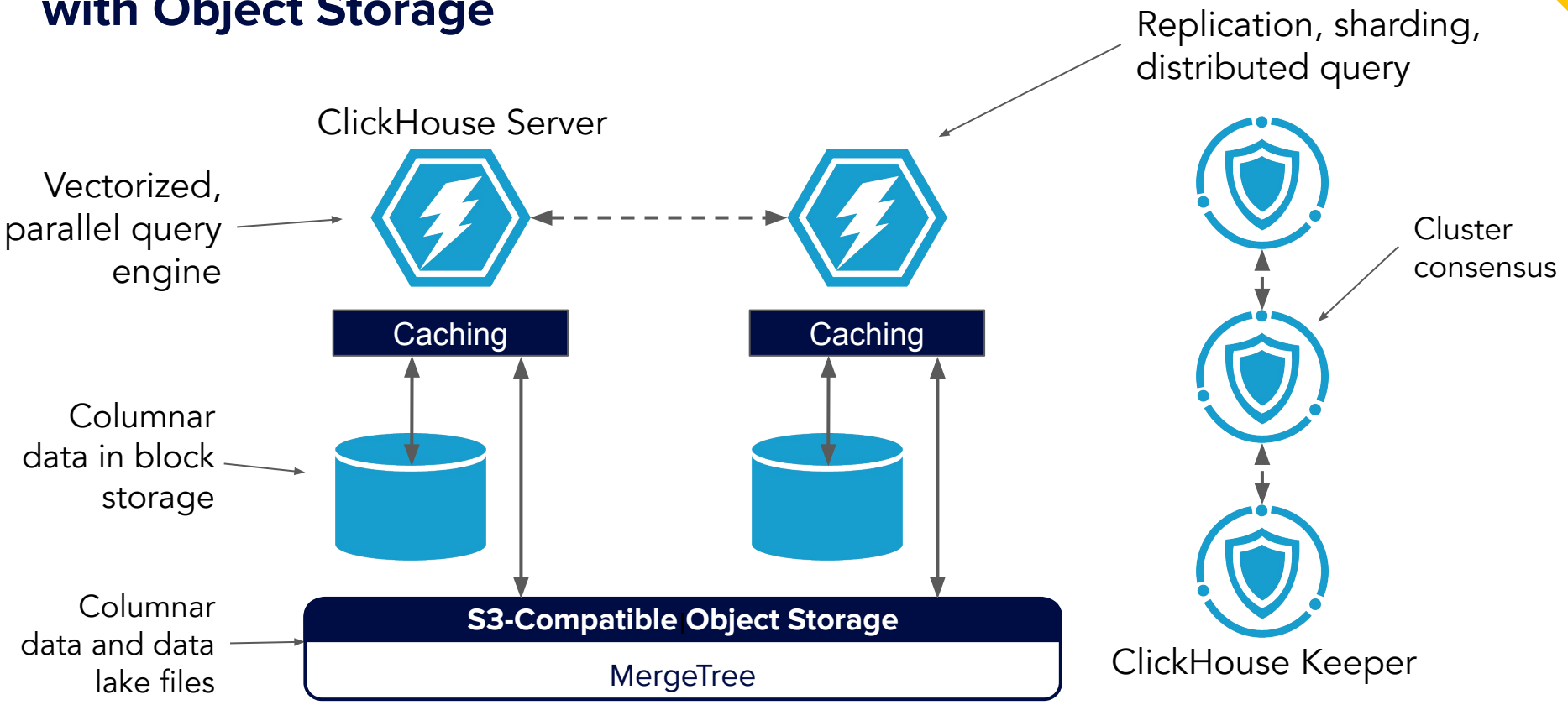
Block storage with replication is 10x more expensive




# Users

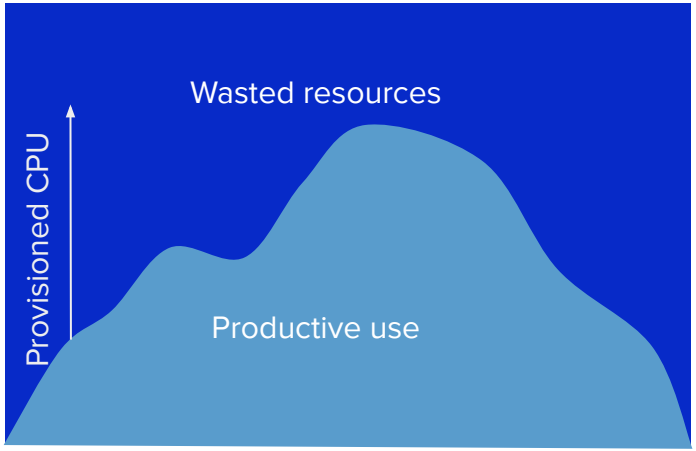
We need object storage to keep PBs of data effectively, or we will go to somewhere else

# ClickHouse shared nothing architecture with Object Storage



# The Next Problem

Overprovisioning wastes compute




# Users

We need the true compute storage separation and scale compute on demand.

And we also need to read this data not only by ClickHouse

# Why is ClickHouse so fast? Round up the usual suspects...

Codecs

Column Storage

Optimized Algorithms

Adaptive Hash Structures

Data Partitioning

Compression

Vectorized Execution

In-RAM dictionaries

Skip Indexes

Projections

Distributed Query

Primary key indexes

# Why is ClickHouse so fast? Round up the usual suspects...

Codecs

Optimized Algorithms

**Users: whatever you do, keep ClickHouse speed!**

Storage

Hash Structures

Data Partitioning

Compression

Skip Indexes

Projections

Vectorized Execution

Distributed Query

In-RAM dictionaries

Primary key indexes

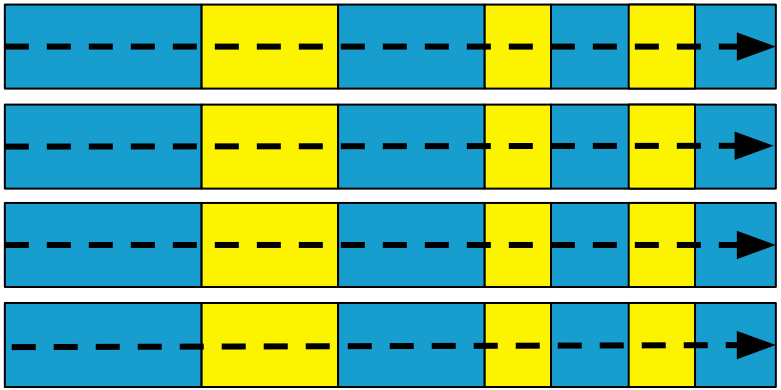
# Goals:

- Object Storage
- Shareable Data Format
- Compute-storage separation
- Real-time ClickHouse speed

# ClickHouse stores table data in compressed columns

## PostgreSQL, MySQL

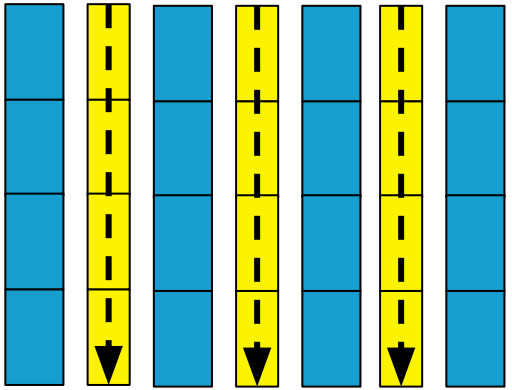
Read all columns in row



Rows minimally or not compressed

## ClickHouse

Read only selected columns



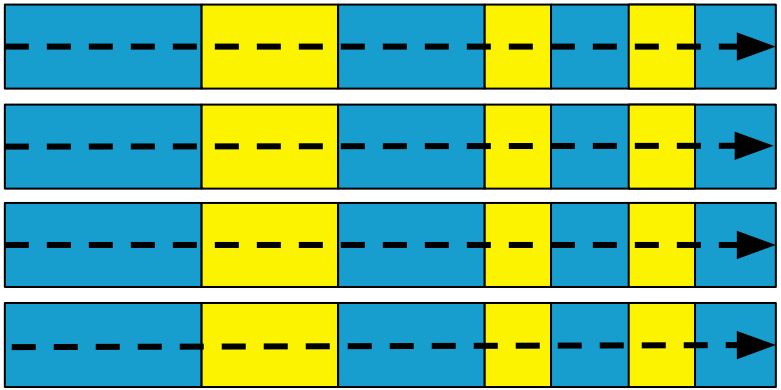
Columns highly compressed

# Parquet

## ~~ClickHouse~~ stores table data in compressed columns

PostgreSQL, MySQL

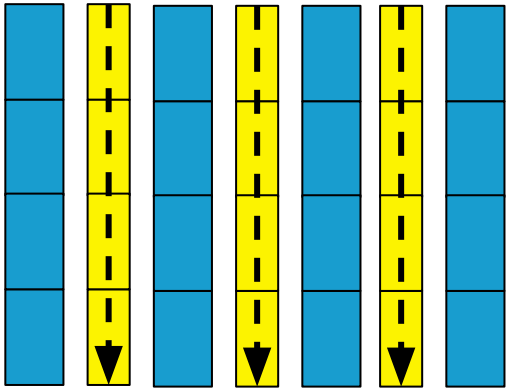
Read all columns in row



Rows minimally or not compressed

Parquet

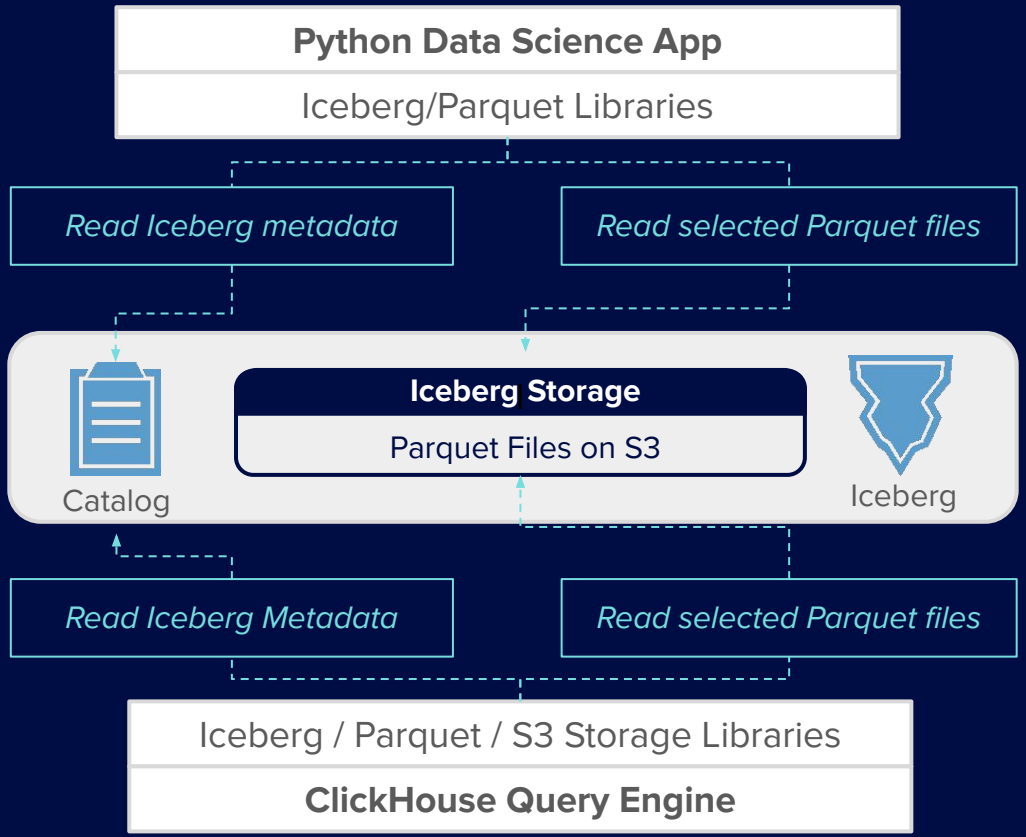
Read only selected columns



Columns highly compressed

# Apache Iceberg

- Organizes files on S3 into tables
- Catalog for table metadata
- Many apps can read same data
- 100% open source, widely adopted
- Not a DBMS!



# Parquet + Iceberg

- ✓ Cheap object storage
- ✓ Columnar
- ✓ Partitioning and partition pruning
- ✓ Ordering and skipping based on ordering
- ✓ Bloom filters
- ✓ Schema evolution
- ✓ Extensible

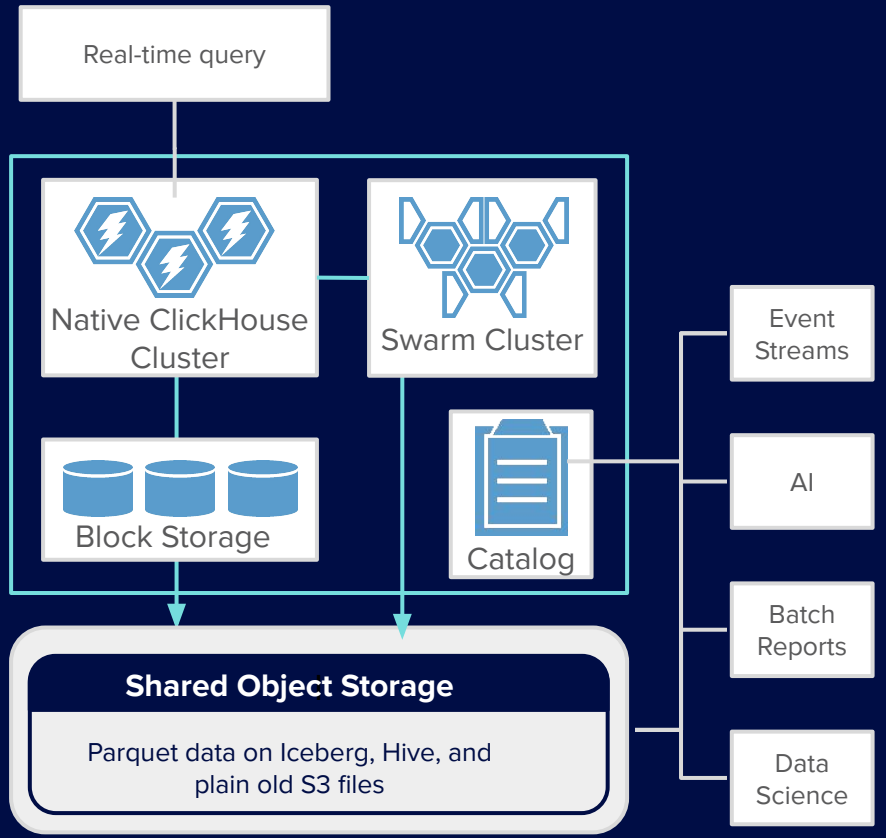
**...but NO query engine...**

# Goals:

- Object Storage
- Shareable Data Format
- Compute-storage separation
- Real-time ClickHouse speed

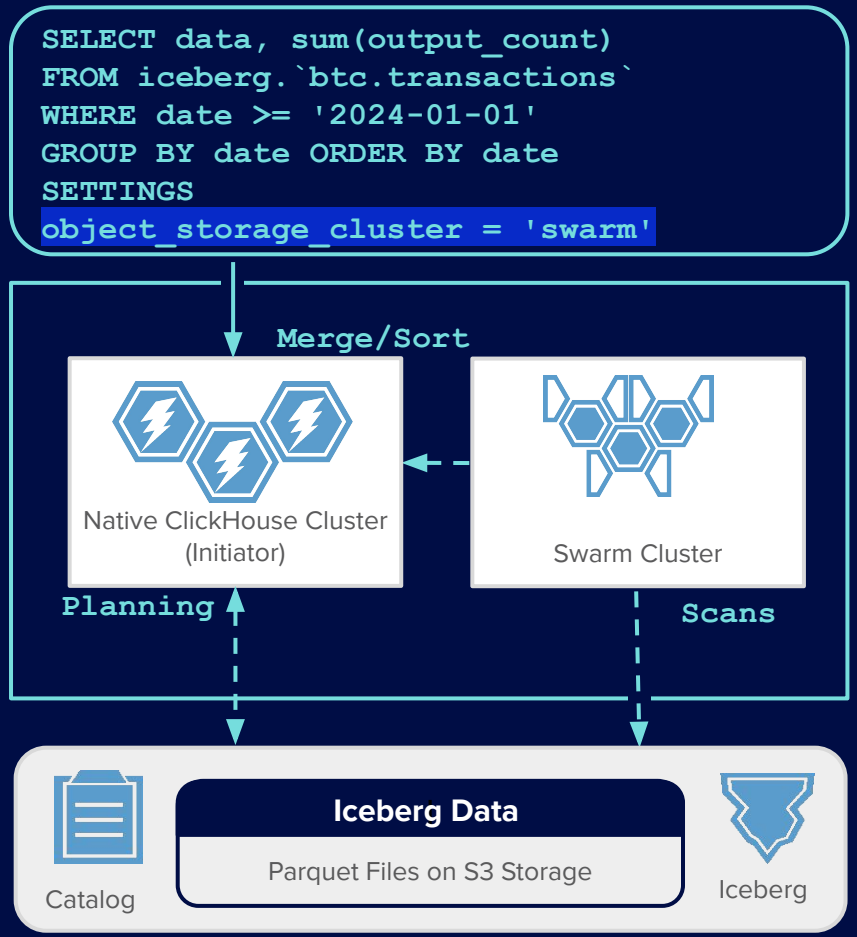
# Project Antalya: shared storage on Iceberg and ClickHouse query engine

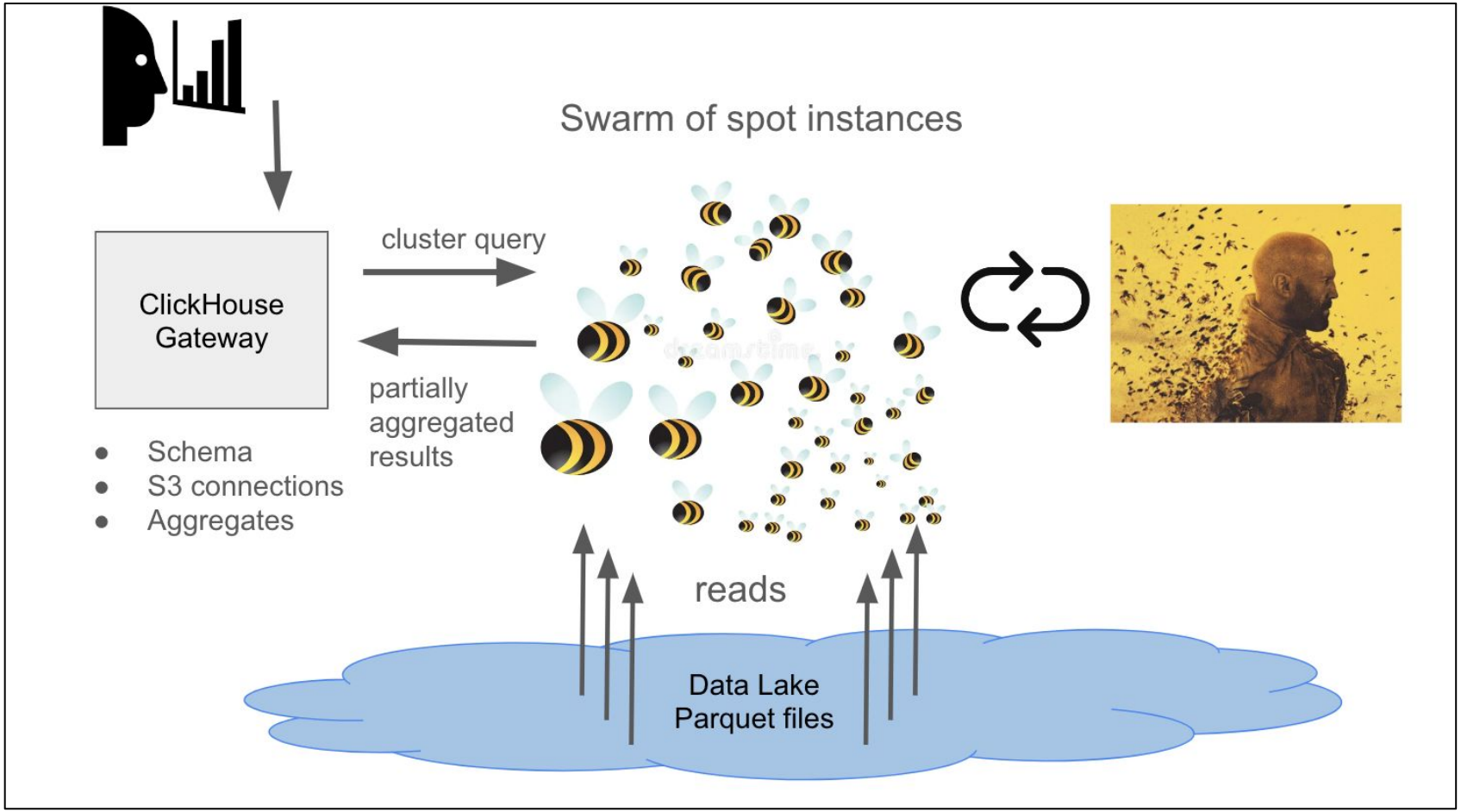
- Extends native ClickHouse capabilities
- Adds Iceberg for shared storage
- Adds swarm clusters for scalable compute
- 100% open source



# Project Antalya query model

- Initiator plans the query
- Swarm Cluster nodes scan shared files
- Initiator merges and sorts responses





# Queries directly on Hive tables on S3 using swarm cluster

```
SELECT date, sum(output_count)
FROM s3('s3://aws-public-blockchain/v1.0/btc/transactions/**.parquet',
  NOSIGN)
WHERE date >= '2025-01-01'
GROUP BY date ORDER BY date ASC
SETTINGS
  use_hive_partitioning = 1,
  object_storage_cluster = 'swarm',
  object_storage_max_nodes = 2;
```

File paths use hive partitioning

Swarm cluster to use

Use just 2 nodes not all

# Connecting ClickHouse to a REST catalog

```
CREATE DATABASE ice ENGINE =
  DataLakeCatalog('https://iceberg-catalog.your-company.com')
SETTINGS
  catalog_type = 'rest',
  auth_header = 'Authorization: Bearer 0********************************8',
  warehouse = 's3://some-bucket-path';

SHOW TABLES FROM ice;
```

```
    ┌─name──────────────────────────────┐
1.  │ aws-public-blockchain.btc         │
2.  │ aws-public-blockchain.btc_ps_by_date │
    └───────────────────────────────────┘
```

Catalog server URL

Cache Parquet file metadata

Path to Iceberg metadata and data files

# Selecting data from Iceberg tables is way easier

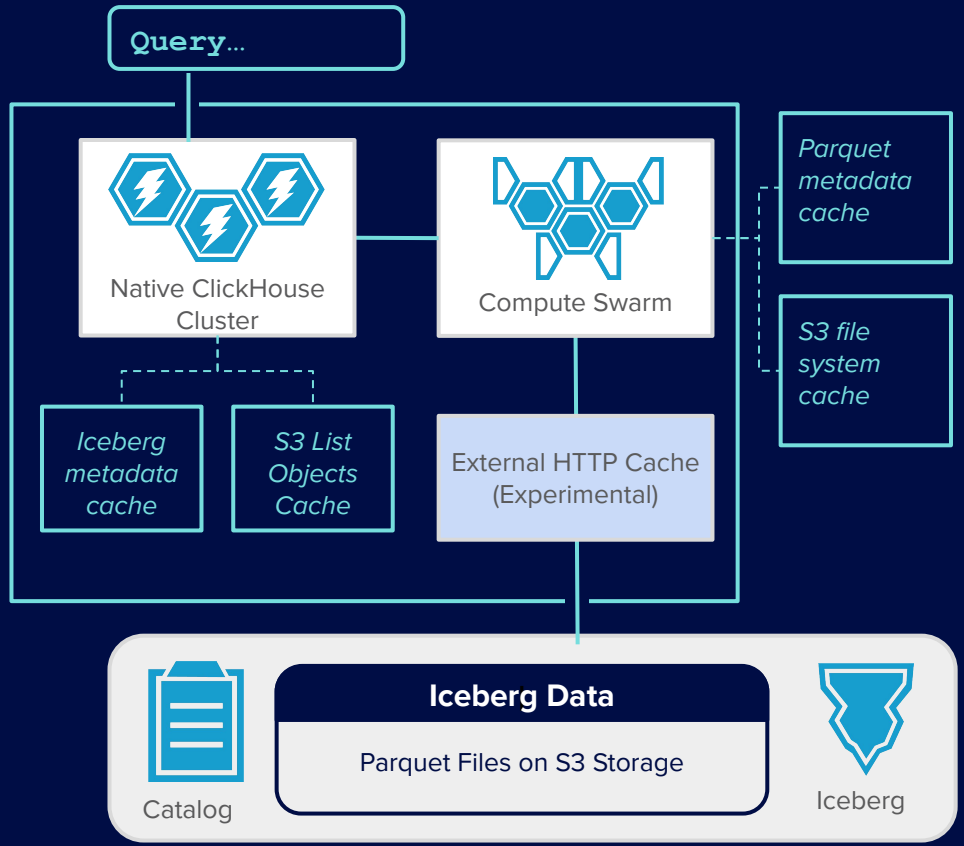
```
SELECT
    date,
    sumDistinct(block_number) AS blocks,
    sum(fee) AS fees,
    fees / blocks AS fee_per_block
FROM ice.`aws-public-blockchain.btc`
WHERE date >= '2025-01-01'
GROUP BY date
ORDER BY date ASC
SETTINGS object_storage_cluster = 'swarm'
```

Backticks required on iceberg table names

Catalog server URL

# More Caching!

- Avoid re-parsing Iceberg metadata
- Avoid re-reading Parquet metadata
- Cache calls to list files in object storage
- Cache fetched S3 blocks on block storage
- Pull fetched S3 blocks from OS Page Cache



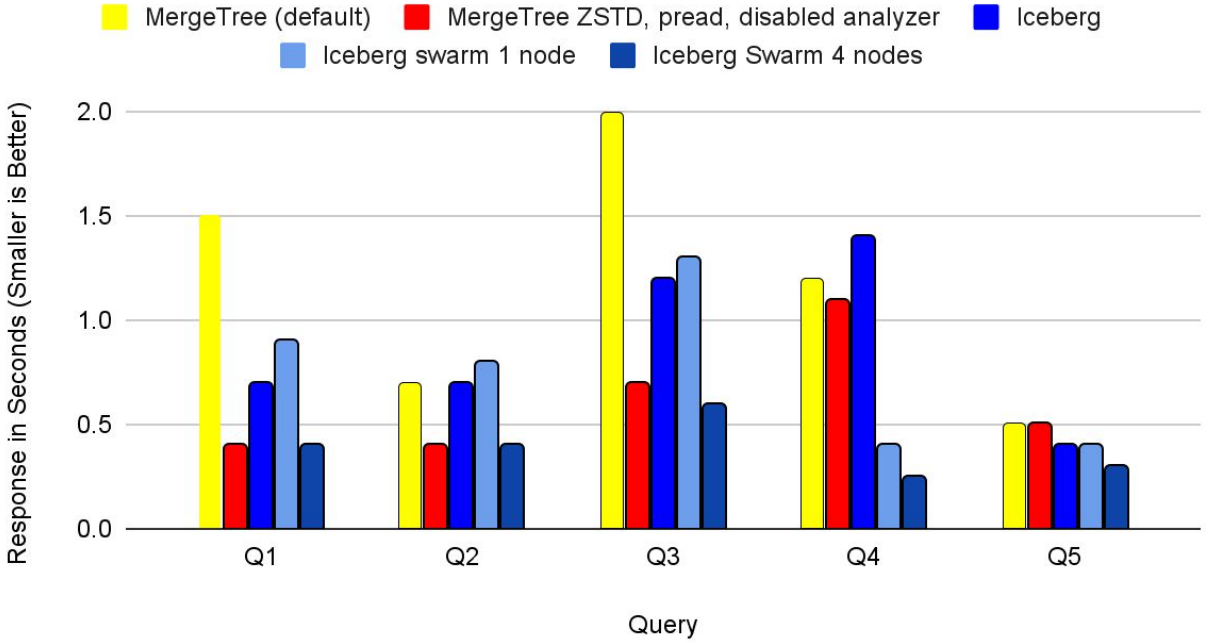
# But is it fast? A simple performance test

- Use nyc.taxis dataset (1.3B rows)
- MergeTree vs. Parquet
- 5 sample queries
- Caches and partition pruning enabled
- AWS c7g.8xlarge with 32vCPUs
- Run 3-5 times, take lowest response

```
Q1:
SELECT
  passenger_count,
  avg(total_amount)
FROM tripdata
GROUP BY
passenger_count
```

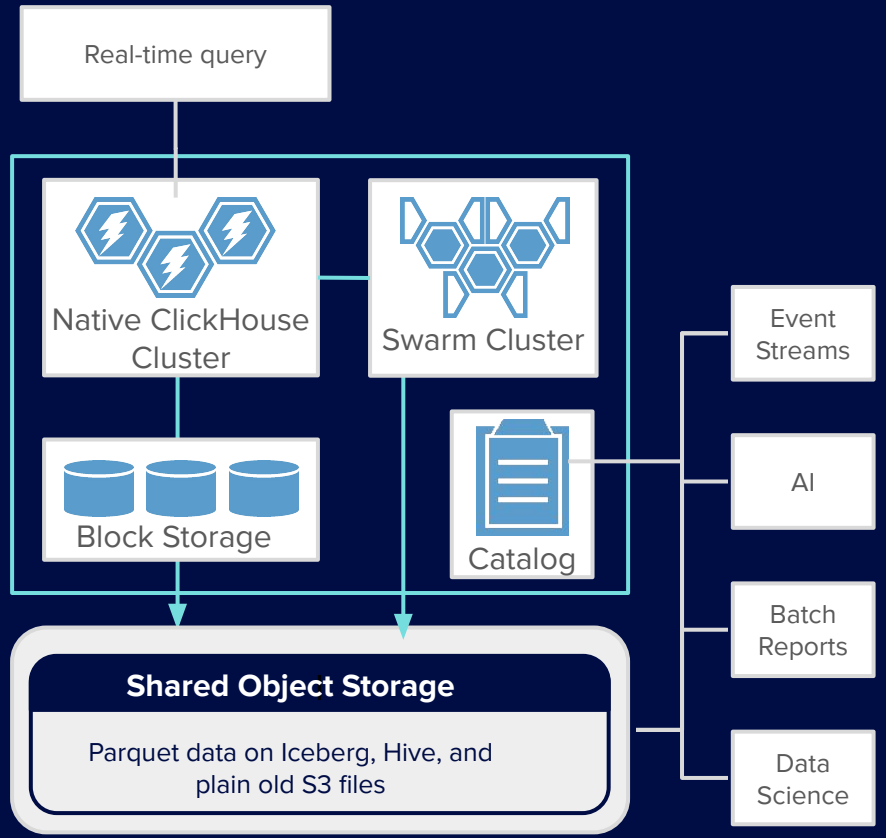
```
Q2:
SELECT
  passenger_count,
  toYear(pickup_date) AS year,
  count(*)
FROM tripdata
GROUP BY passenger_count, year
```

# Parquet and MergeTree response are reaching parity

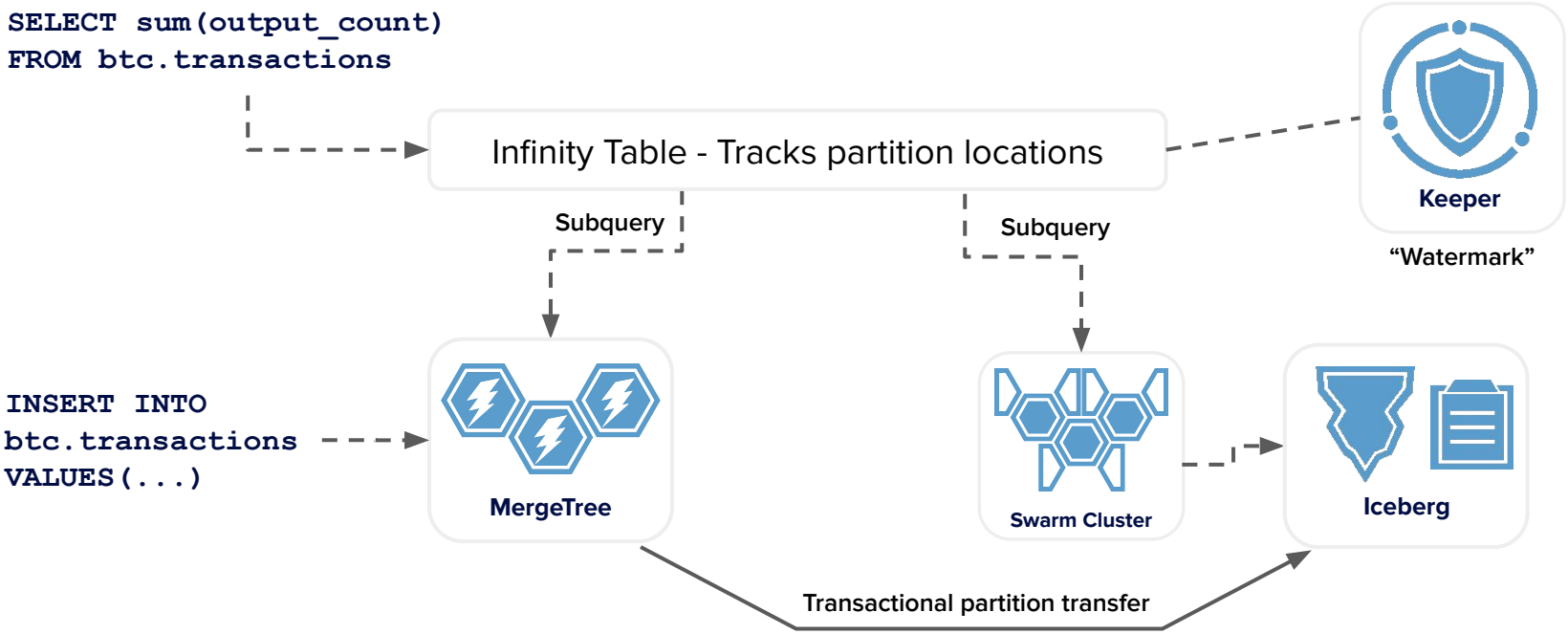


# Iceberg REST Catalog for built-in data lake

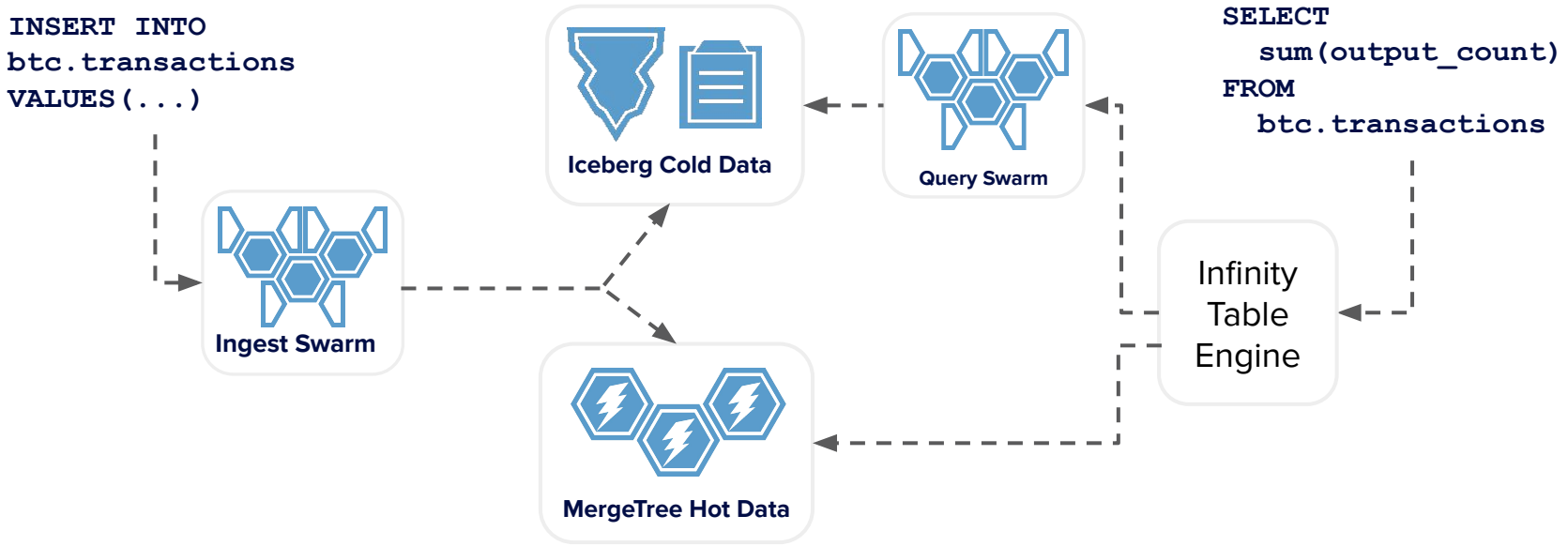
- Wrapper on Iceberg Java SDK
- Simple deployment backed by etcd or SQL database
- Fast, automated loading
- Authentication using bearer tokens
- Compaction support coming
- Supplemented by 3rd party catalog support (AWS Glue, S3 table buckets, etc.)



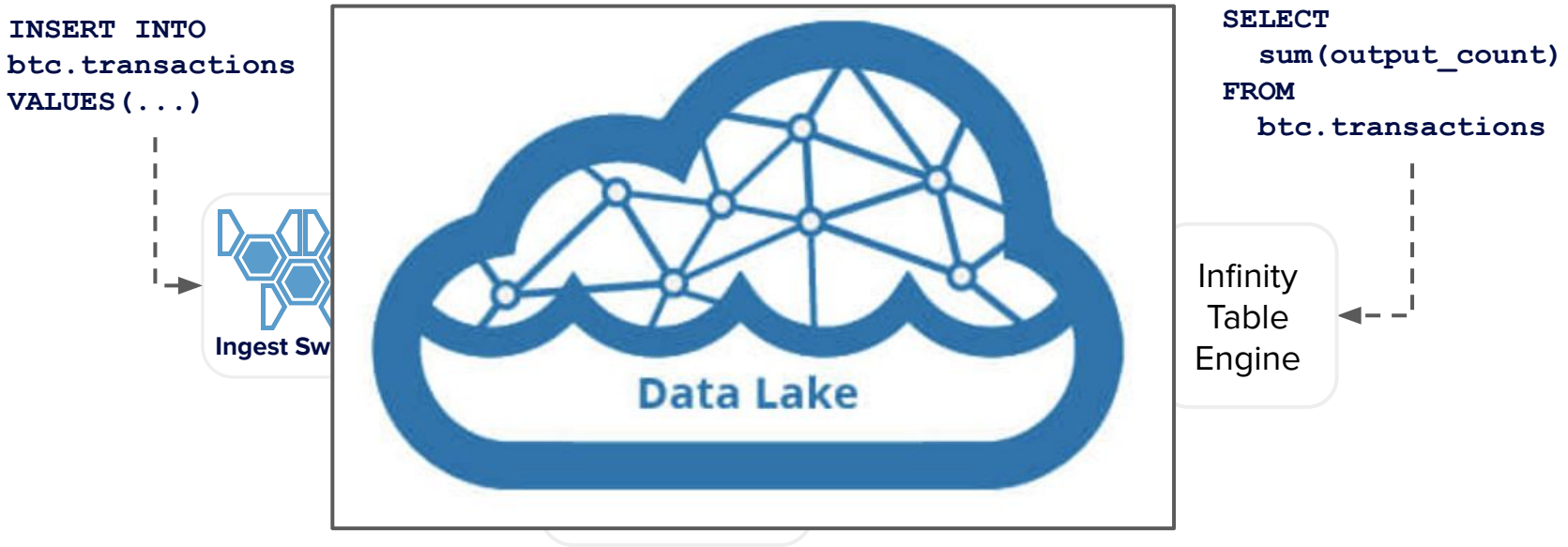
# Infinity Tables – Seamless extension of MergeTree to Iceberg



# 2026 Plans - Double-write architecture for large scale analytics



# 2026 Plans - Real-Time Data Lake Arrives!



# How to use it?

1. Clone the antalya-examples repo.
   ```
   git clone https://github.com/Altinity/antalya-examples
   ```

2. Bring up Kubernetes using Terraform
   ```
   cd antalya-examples/kubernetes/terraform
   terraform init
   terraform apply
   ```

3. Update kubeconfig and start having fun.
   ```
   aws eks update-kubeconfig --name my-eks-cluster
   ```

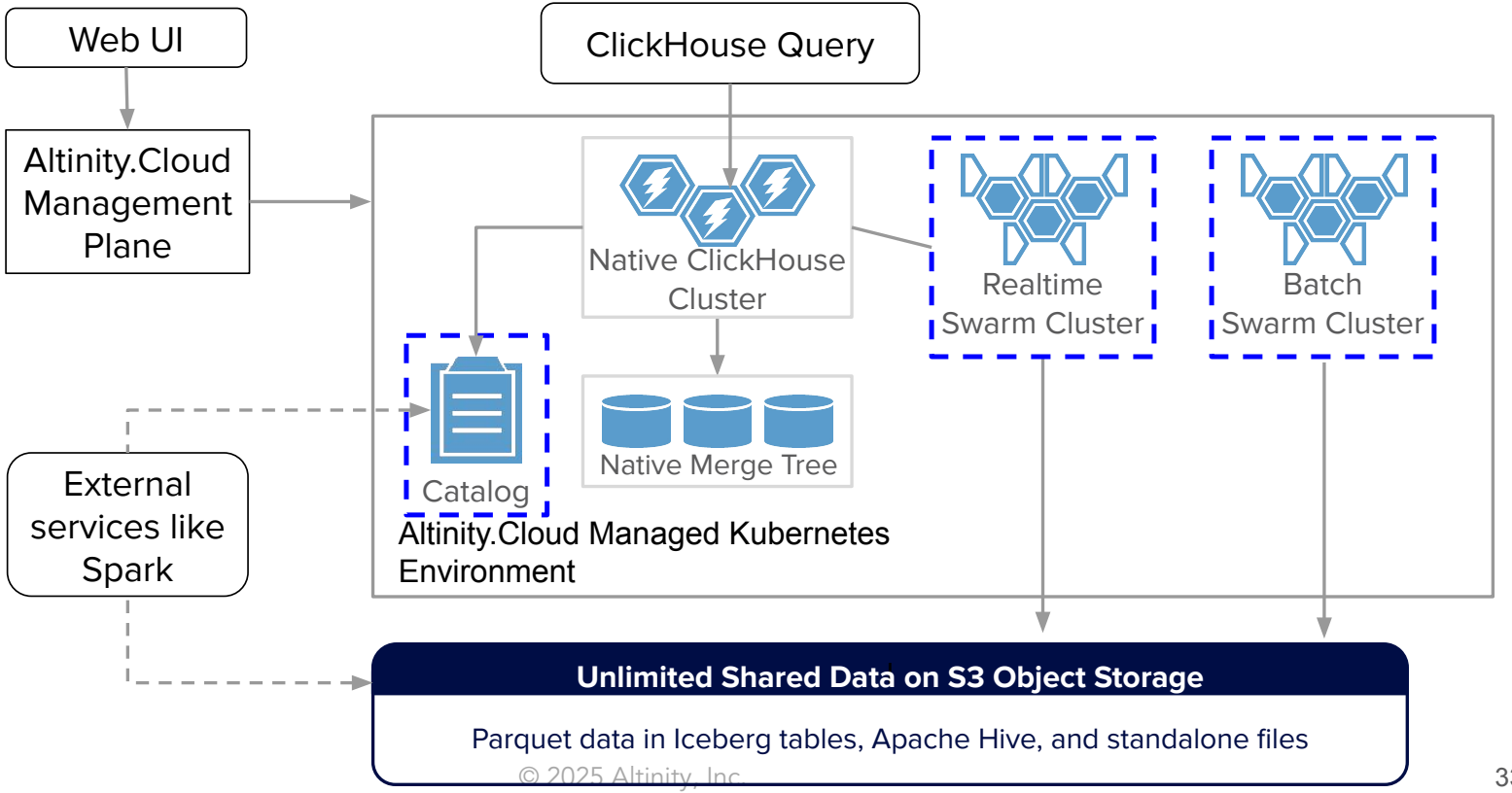
# Bring up Project Antalya with a swarm cluster using manifests

1. Cd to manifests director from terraform directory.
   ```
   cd ../manifests
   ```

2. Load manifests for servers.
   ```
   kubectl apply -f gp3-encrypted-fast-storage-class.yaml
   kubectl apply -f keeper.yaml
   kubectl apply -f swarm.yaml
   kubectl apply -f vector.yaml
   ```

3. Test out your new cluster. This shows the swarm clusters are reachable.
   ```
   kubectl exec -it chi-vector-example-0-0-0 -- clickhouse-client \
   "SELECT hostName(), version() FROM clusterAllReplicas('swarm',
   system.one)"
   ```

# Altinity.Cloud can deploy the Antalya stack anywhere



# Project Antalya resources

- Check out Altinity antalya-examples repo for samples and documentation

  `git@github.com:Altinity/antalya-examples.git`

- Project Antalya code is in the Altinity ClickHouse repo (log issues there)

  `git@github.com:Altinity/ClickHouse.git`

- Read “Getting Started with Altinity’s Project Antalya” to find out more

  `https://altinity.com/blog/getting-started-with-altinitys-project-antalya`

- Join the Altinity Public Slack to find out more: `https://altinity.com/slack`

# Thank you! Questions?

Learn more and join our community

https://altinity.com

Project Antalya

ALTINITY