cloudera®

Introduction to Kudu:
Hadoop Storage for Fast Analytics on Fast Data

Jeff Holoman

Sr. Systems Engineer

Agenda

What is Kudu? (Motivations & Design Goals)

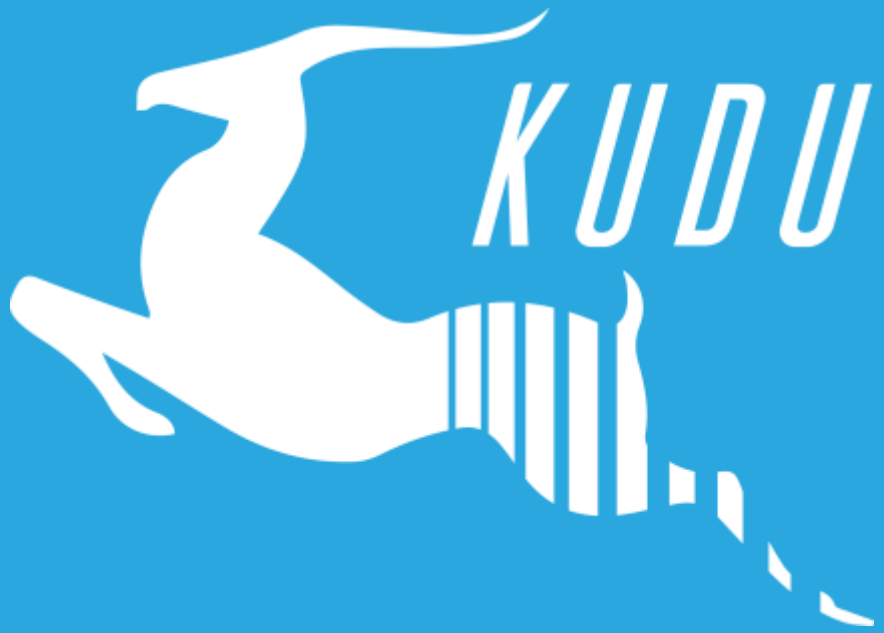
Use Cases

Overview of Design & Internals

Simple Benchmark

Status & Getting Started

cloudera



What is Kudu?

But First….

cloudera

Excels at…

- Efficiently scanning large amounts of data
- Accumulating data with high throughput
- Multiple SQL Options
- All processing engines

However…

- Single Row Access is problematic
- Mutation is problematic
- “Fast Data” access is problematic

GFS paper published in 2003!

Excels at…

- Efficiently finding and writing individual rows
- Accumulating data with high throughput

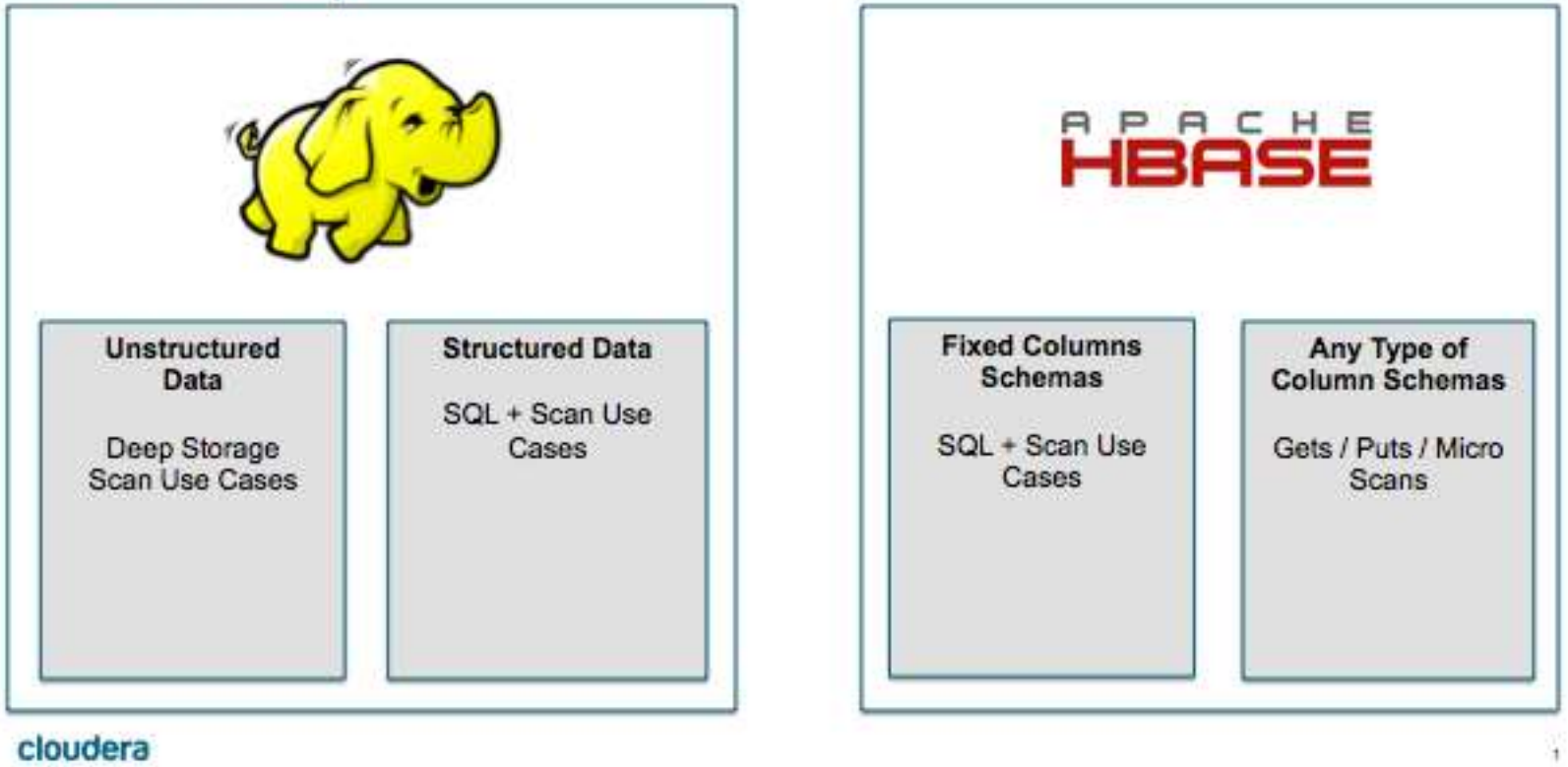
However…

- Scans are problematic
- High cardinality access is problematic
- SQL support is so/so due to the above

Big Table Paper published in 2006!

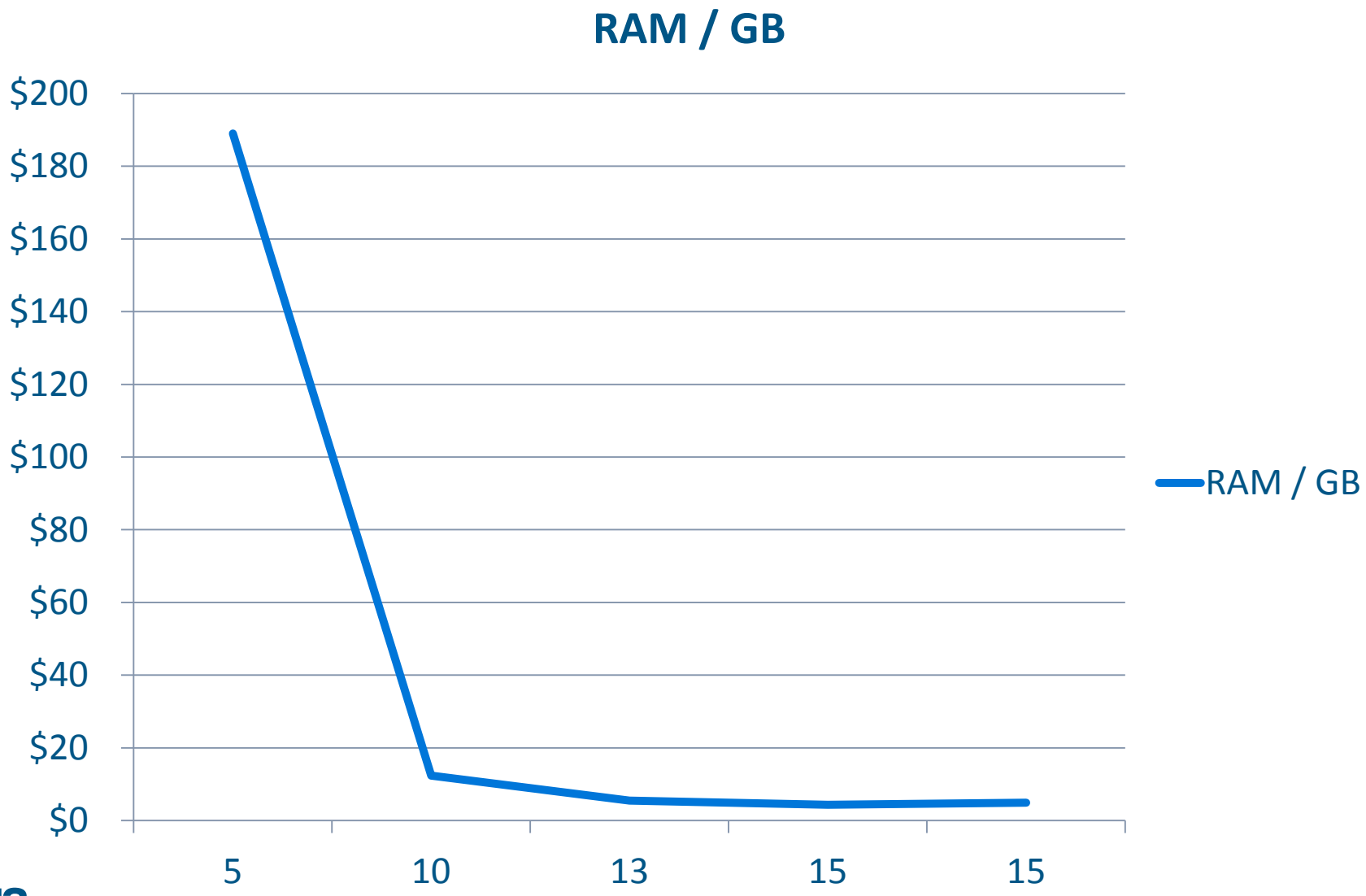
New Hadoop Storage Option

Use case break up

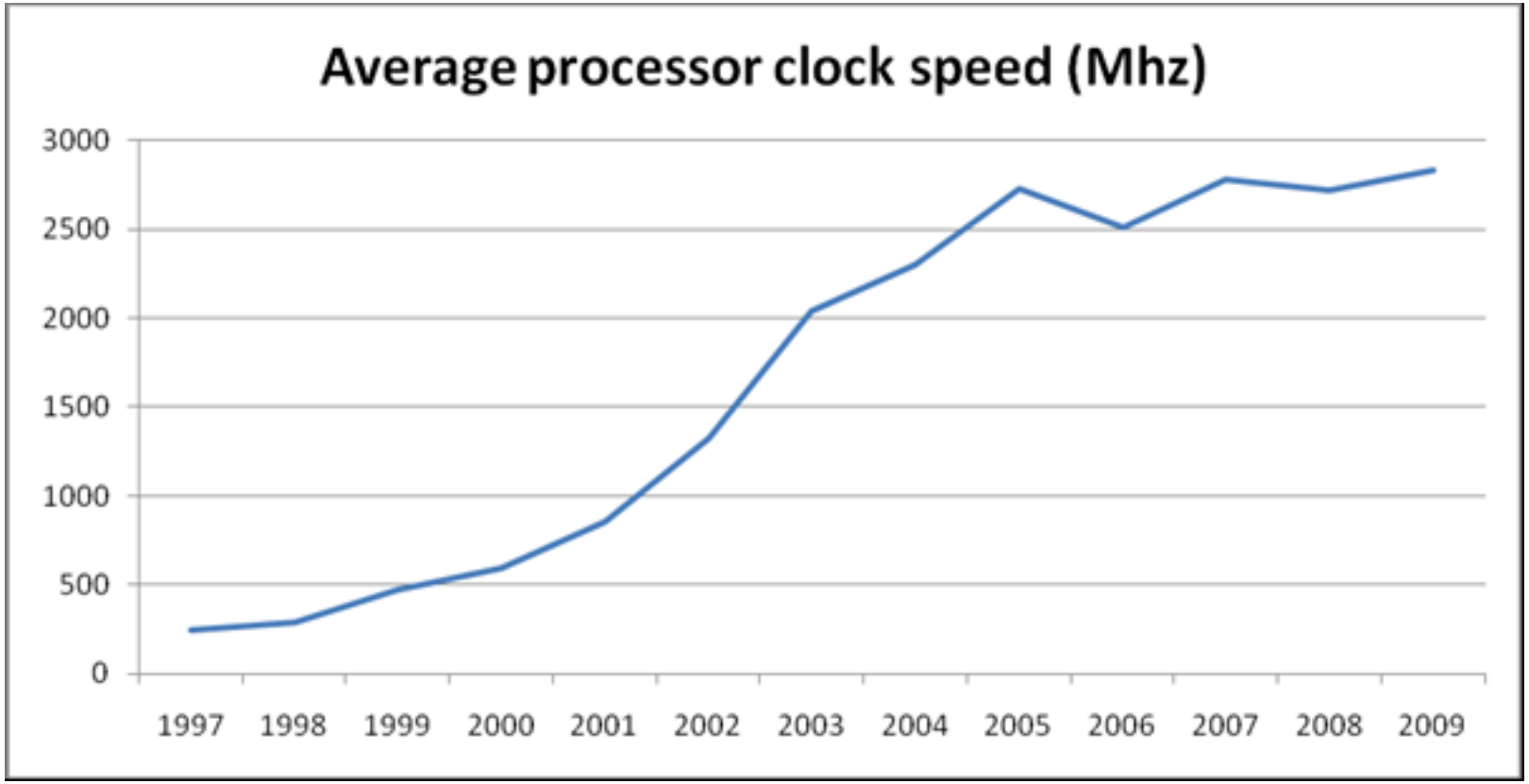


In 2006…

cloudera



cloudera



cloudera

Today…Changing Hardware Landscape

- **Spinning disk** -> **solid state storage**
  - **NAND flash**: Up to 450k read 250k write iops, about 2GB/sec read and 1.5GB/sec write throughput, at a price of less than $3/GB and dropping
  - **3D XPoint memory** (1000x faster than NAND, cheaper than RAM)

- **RAM is cheaper and more abundant**
  - 64->128->256GB over last few years

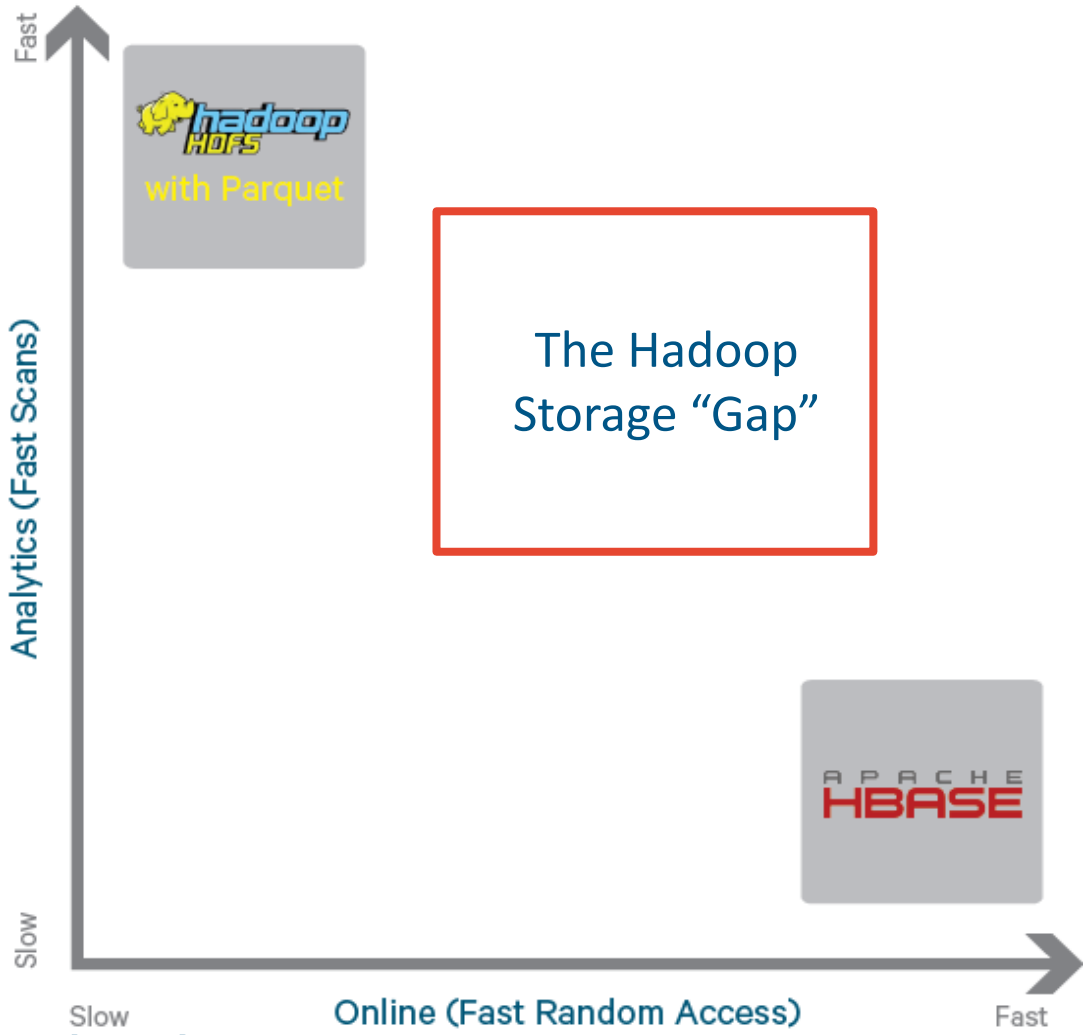
***Takeaway 1*:**

The **next bottleneck is CPU**, and current storage systems weren't designed with CPU efficiency in mind.

Takeaway 2:

Column stores are feasible for random access.

Current Storage Landscape in Hadoop

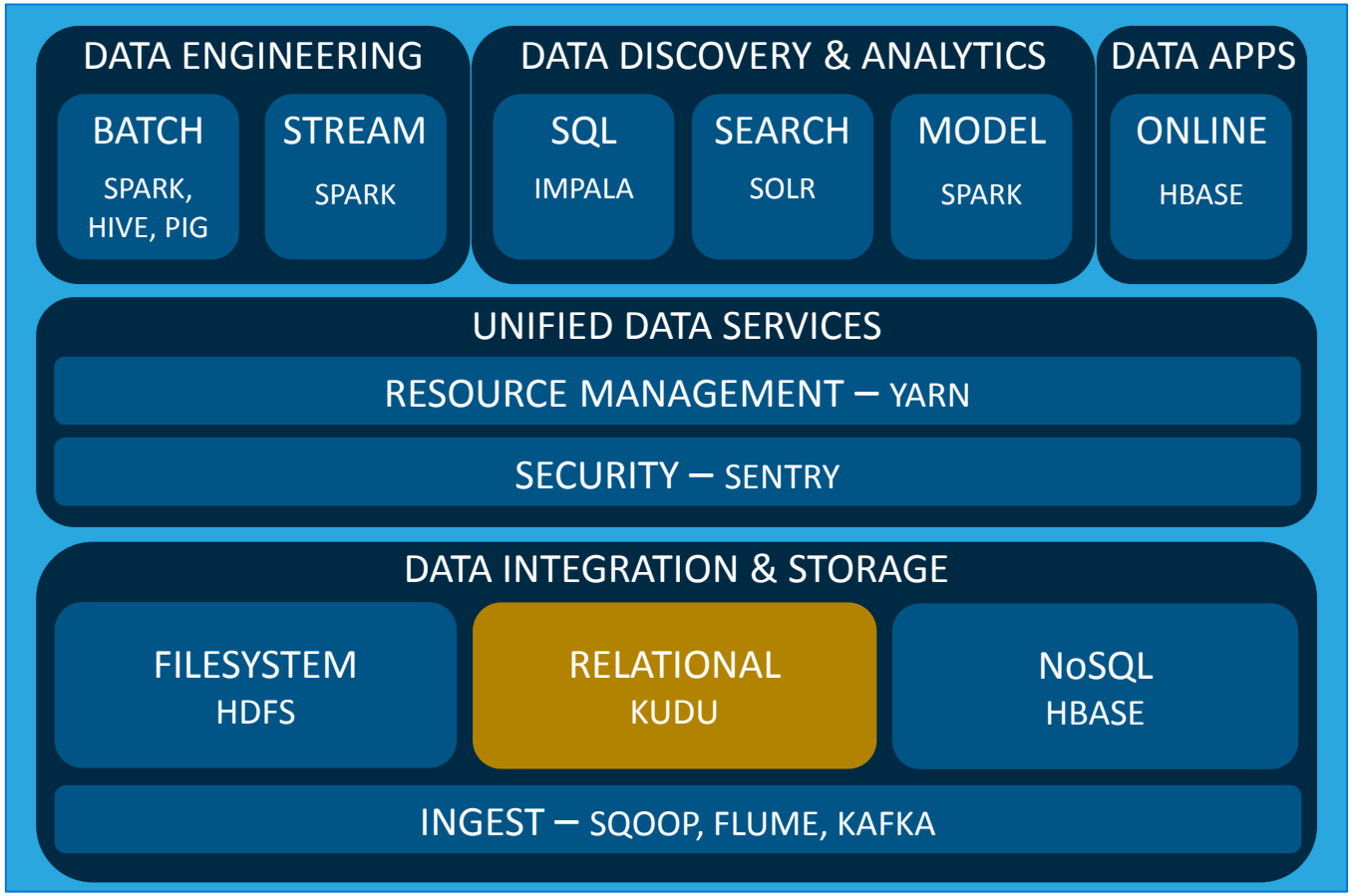


Gaps exist when these properties are needed *simultaneously*

cloudera

The Kudu Elevator Pitch

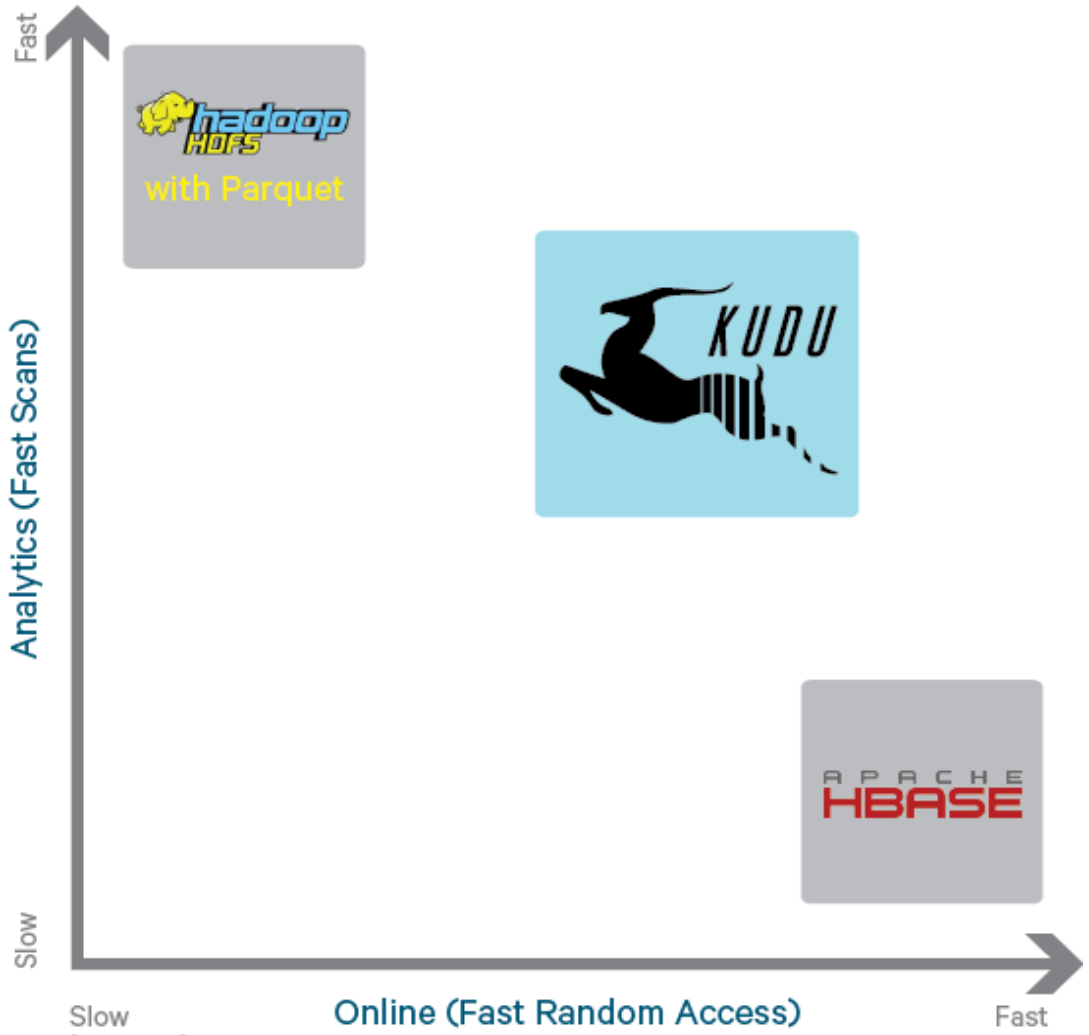
Storage for Fast Analytics on Fast Data



- New updating column store for Hadoop
  - Simplifies the architecture for building analytic applications on changing data
  - Designed for fast analytic performance
  - Natively integrated with Hadoop
- Apache-licensed open source (with pending ASF Incubator proposal)
- Beta now available

cloudera

Kudu Design Goals

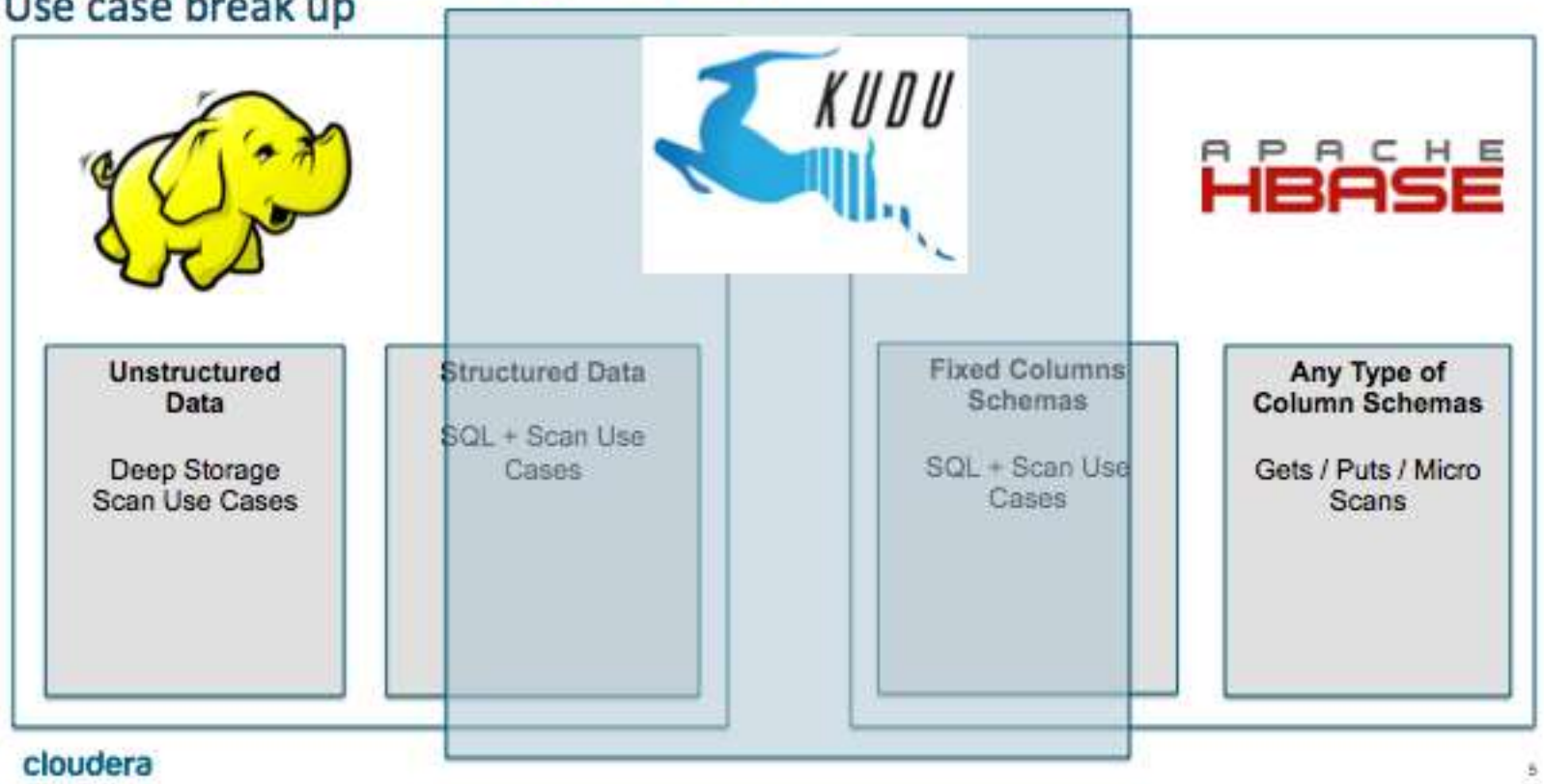


- **High throughput** for big scans
- **Low-latency** for random accesses
- **High CPU performance** to better take advantage of RAM and Flash
  - Single-column scan rate 10-100x faster than HBase
- **High IO efficiency**
  - True column store with type-specific encodings
  - Efficient analytics when only certain columns are accessed
- **Expressive** and **evolvable** data model
- Architecture that supports **multi-data center operation**

cloudera

New Hadoop Storage Option

Use case break up

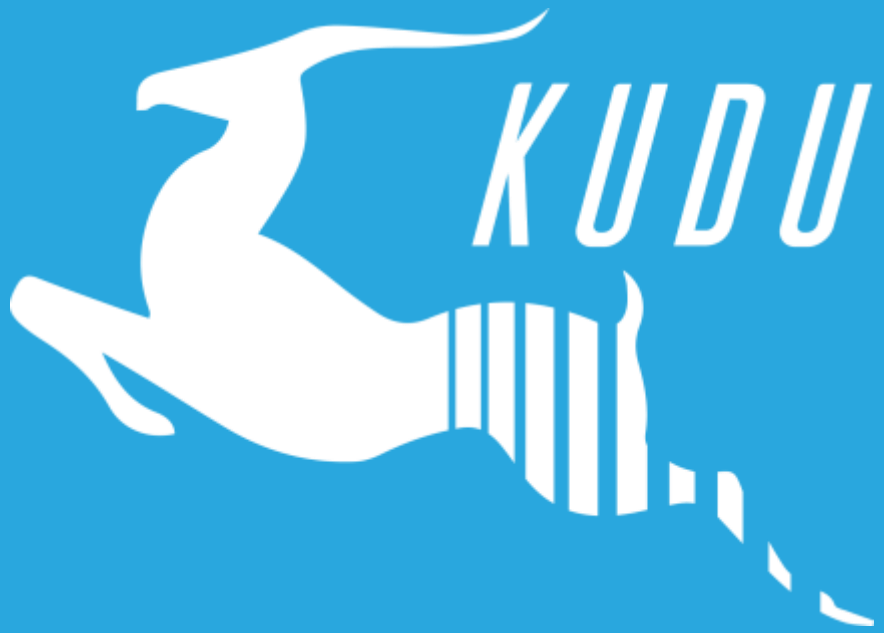


Using Kudu

- Table has a **SQL-like schema**
  - **Finite number of columns** (unlike HBase/Cassandra)
  - **Types**: BOOL, INT8, INT16, INT32, INT64, FLOAT, DOUBLE, STRING, BINARY, TIMESTAMP
  - Some subset of columns makes up a **possibly-composite primary key**
  - Fast ALTER TABLE
- Java and C++ **"NoSQL" style APIs**
  - Insert(), Update(), Delete(), Scan()
- Integrations with **MapReduce, Spark, and Impala**
  - more to come!

What Kudu is *NOT*

- *Not* a SQL interface itself
  - It's just the storage layer – "Bring Your Own SQL" (eg Impala or Spark)
- *Not* an application that runs on HDFS
  - It's an alternative, native Hadoop storage engine
  - Colocation with HDFS is expected
- *Not* a replacement for HDFS or HBase
  - Select the right storage for the right use case
  - Cloudera will support and invest in all three



Use Cases for Kudu

Kudu Use Cases

Kudu is best for use cases requiring a simultaneous combination of sequential and random reads and writes, e.g.:

- **Time Series**
  - Examples: Stream market data; fraud detection & prevention; risk monitoring
  - Workload: Insert, updates, scans, lookups
- **Machine Data Analytics**
  - Examples: Network threat detection
  - Workload: Inserts, scans, lookups
- **Online Reporting**
  - Examples: ODS
  - Workload: Inserts, updates, scans, lookups

Industry Examples

Financial Services

- Streaming market data
- Real-time fraud detection & prevention
- Risk monitoring

Retail

- Real-time offers
- Location-based targeting

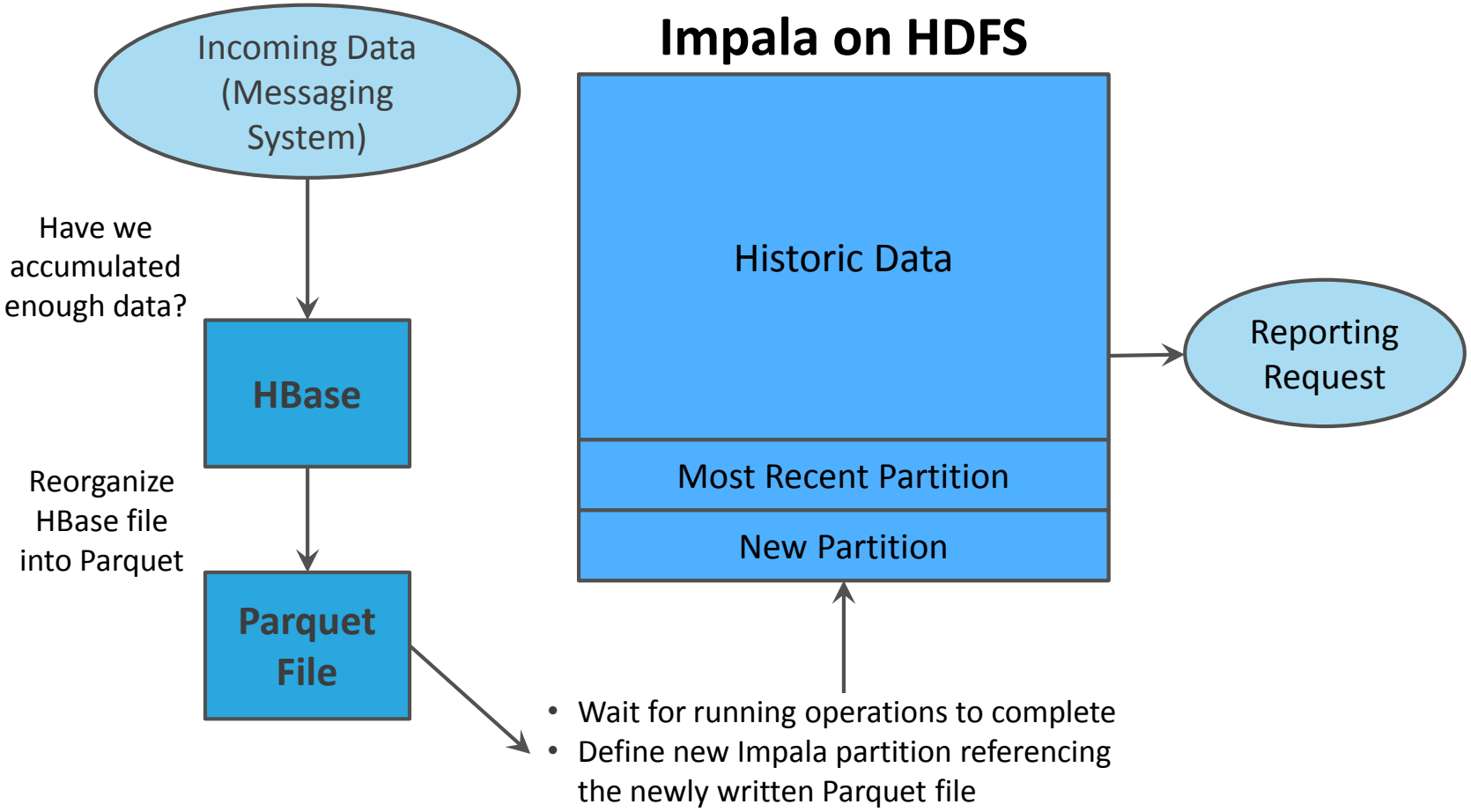
Public Sector

- Geospatial monitoring
- Risk and threat detection (real time)

cloudera

Real-Time Analytics in Hadoop Today

Fraud Detection in the Real World = Storage Complexity



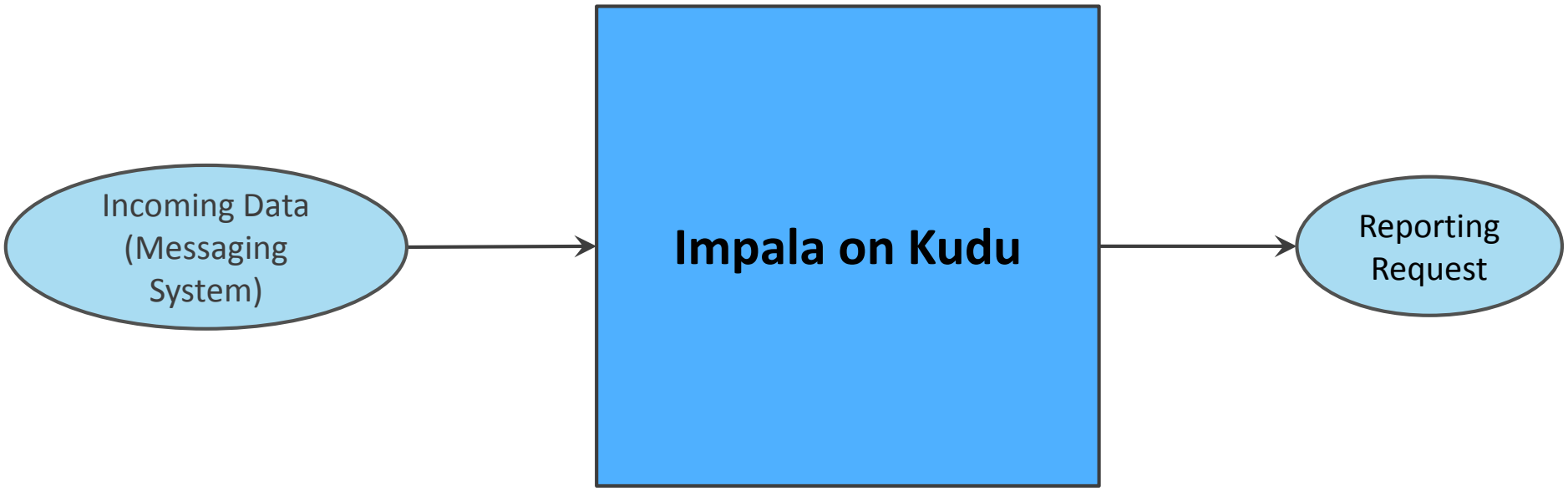
Considerations:

- How do I handle failure during this process?
- How often do I reorganize data streaming in into a format appropriate for reporting?
- When reporting, how do I see data that has not yet been reorganized?
- How do I ensure that important jobs aren't interrupted by maintenance?

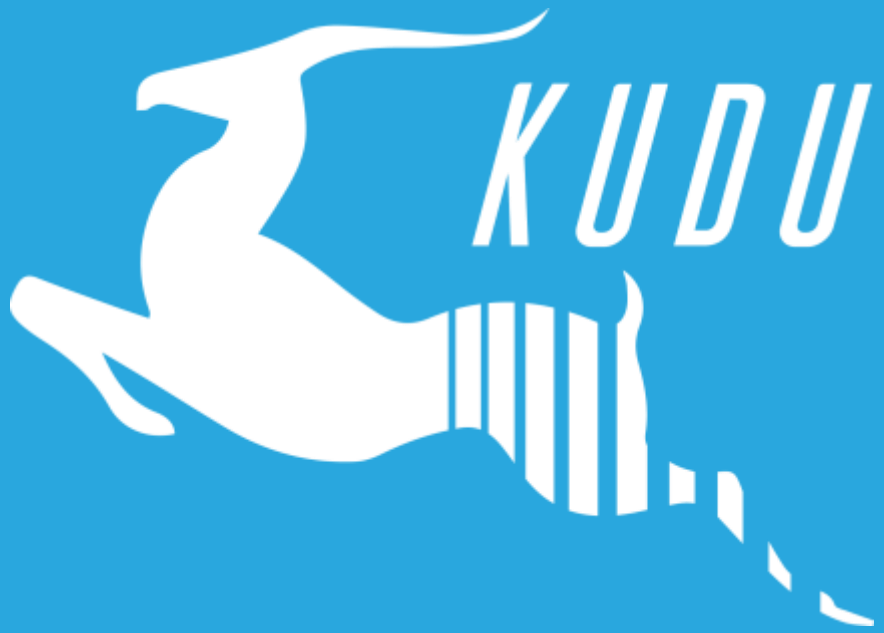
cloudera

Real-Time Analytics in Hadoop with Kudu

Simpler Architecture, Superior Performance over Hybrid Approaches



cloudera



Design & Internals

Kudu Basic Design

- Typed storage
- Basic Construct: *Tables*
  - Tables broken down into *Tablets* (roughly equivalent to partitions)
- Maintains consistency through a Paxos-like quorum model (Raft)
- Architecture supports geographically disparate, active/active systems

Columnar Data Store

A	B	C
A1	B1	C1
A2	B2	C2
A3	B3	C3

Row-Based Storage

A1	B1	C1	A2	B2	C2	A3	B3	C3

Columnar Storage

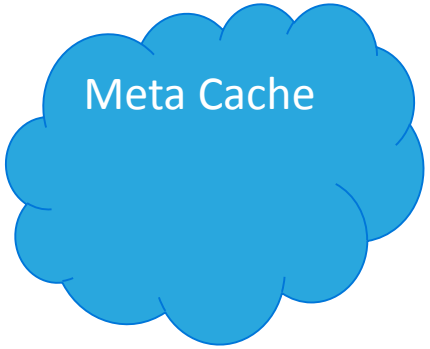
A1	A2	A3	B1	B2	B3	C1	C2	C3

Tables and Tablets

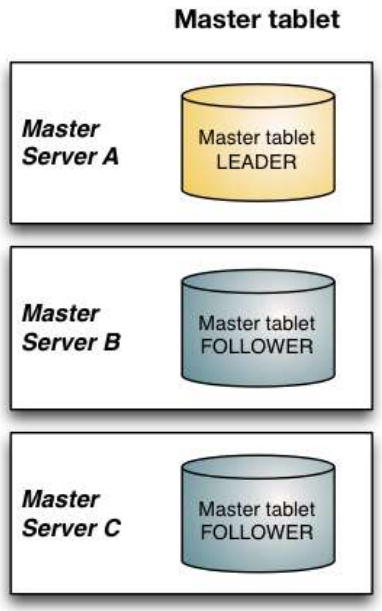
- Table is **horizontally partitioned into *tablets***
  - *Range* or *hash* partitioning
  - `PRIMARY KEY (host, metric, timestamp) DISTRIBUTE BY HASH(timestamp) INTO 100 BUCKETS`
- Each tablet has N replicas (3 or 5), with **Raft consensus**
  - Allow read from any replica, plus leader-driven writes with low MTTR
- **Tablet servers** host tablets
  - Store data on **local disks** (no HDFS)

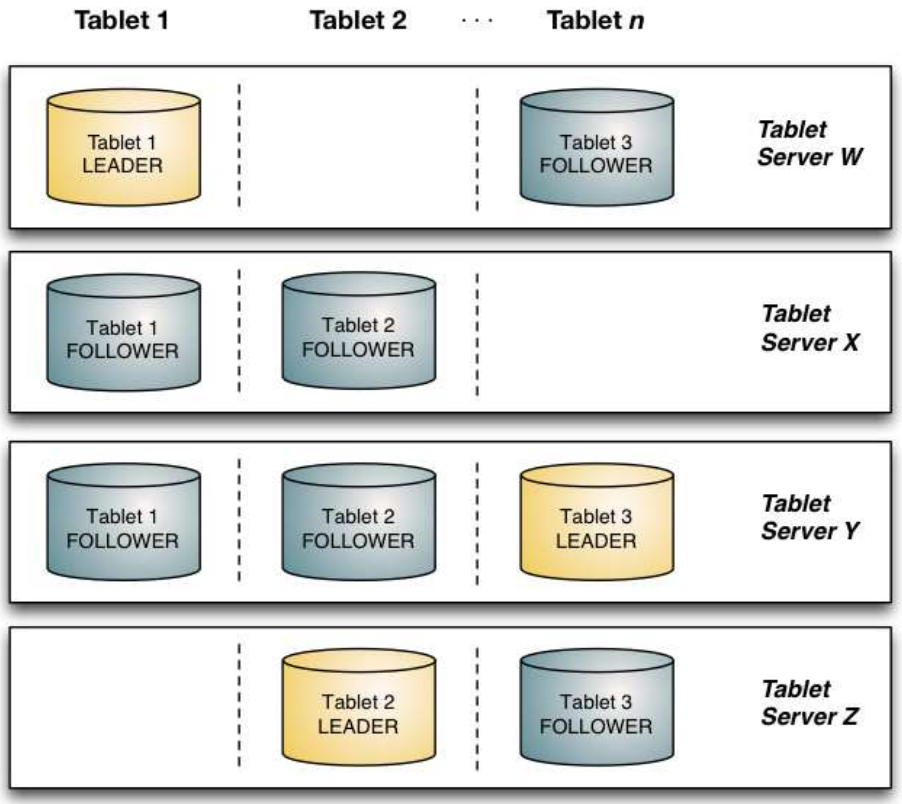
Tables and Tablets(2)

- CREATE TABLE customers (
- first_name STRING NOT NULL,
- last_name STRING NOT NULL,
- order_count INT32,
- PRIMARY KEY (last_name, first_name),
-)
- Specifying the split rows as (("b", ""), ("c", ""), ("d", ""), .., ("z", "")) (25 split rows total) will result in the creation of 26 tablets, with each tablet containing a range of customer surnames all beginning with a given letter. This is an effective partition schema for a workload where customers are inserted and updated uniformly by last name, and scans are typically performed over a range of surnames.

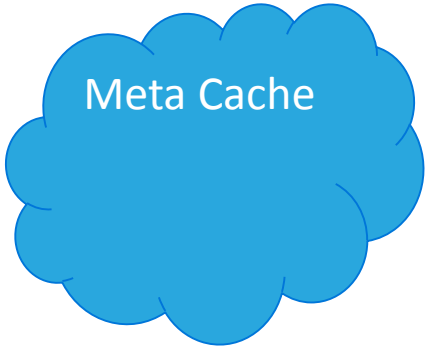


Client



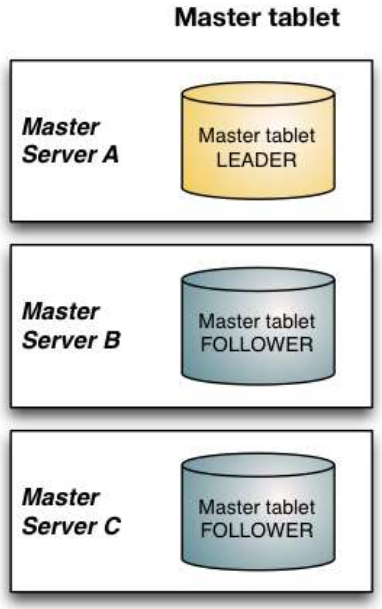


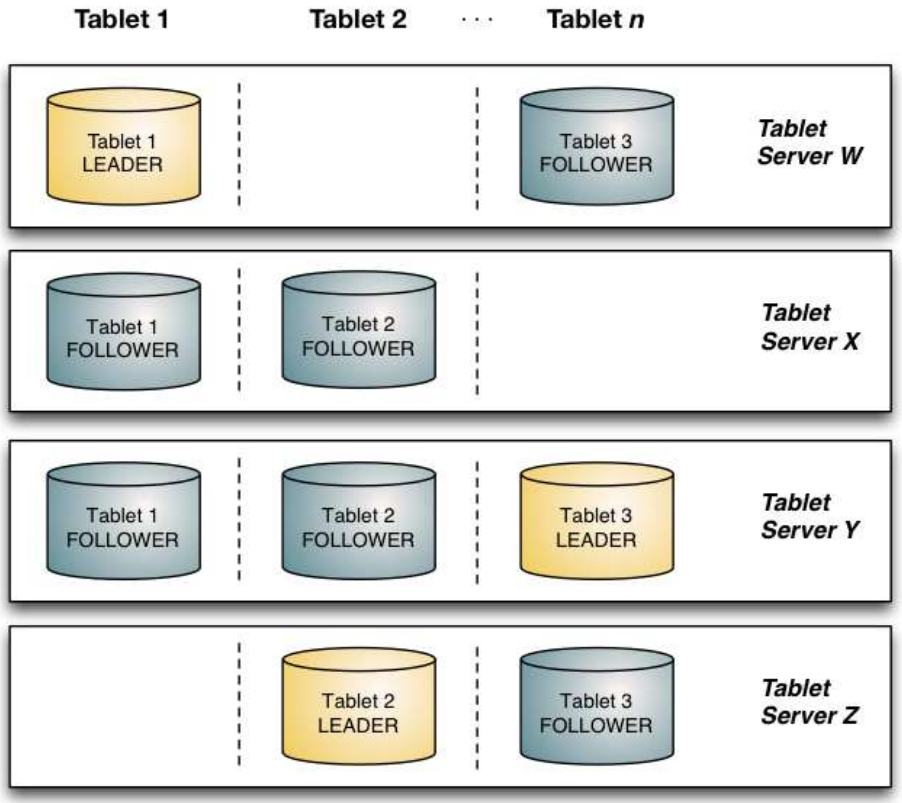
cloudera



Client

Hey Master! Where is the row for 'todd@cloudera.com' in table "T"?





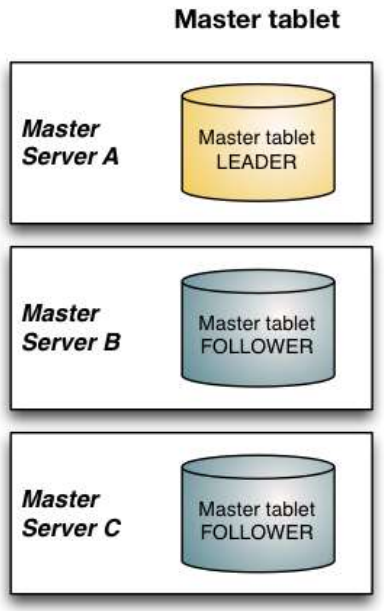
cloudera

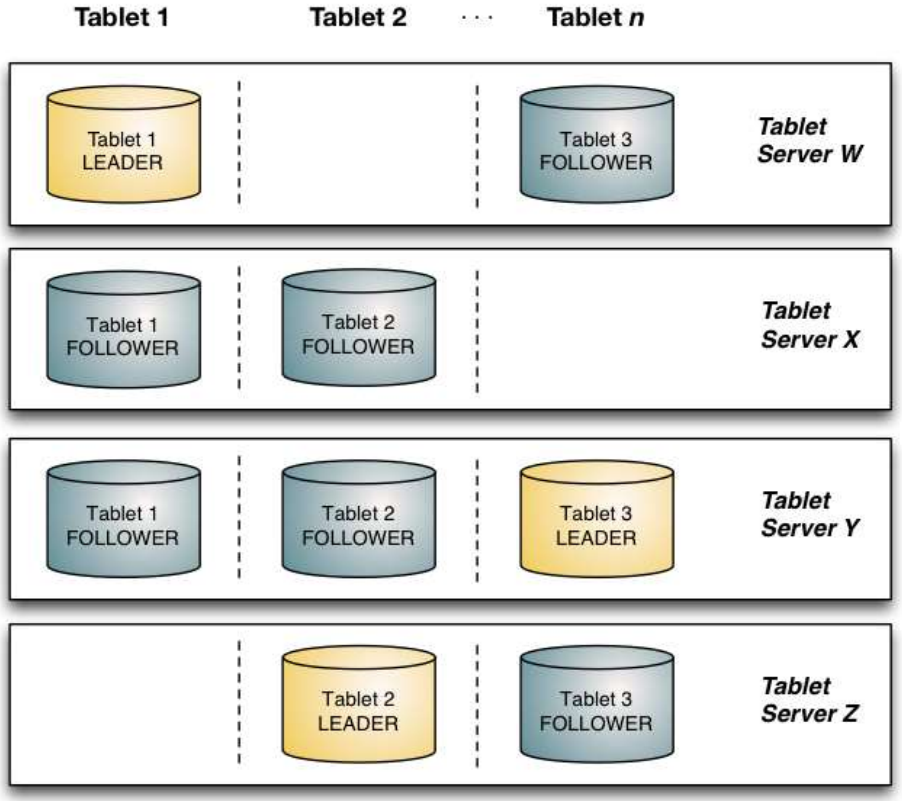
Meta Cache

Client

Hey Master! Where is the row for 'todd@cloudera.com' in table "T"?

It's part of tablet 2, which is on servers {Z,Y,X}. BTW, here's info on other tablets you might care about: T1, T2, T3, …





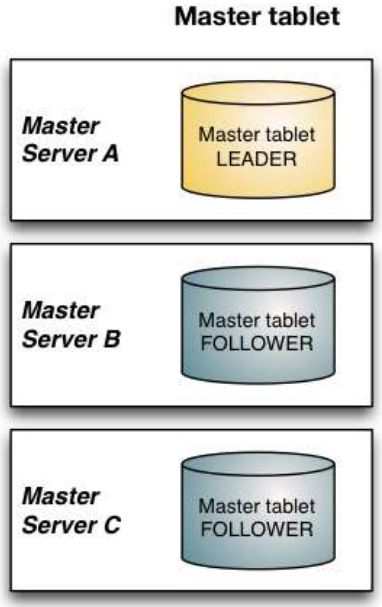
cloudera

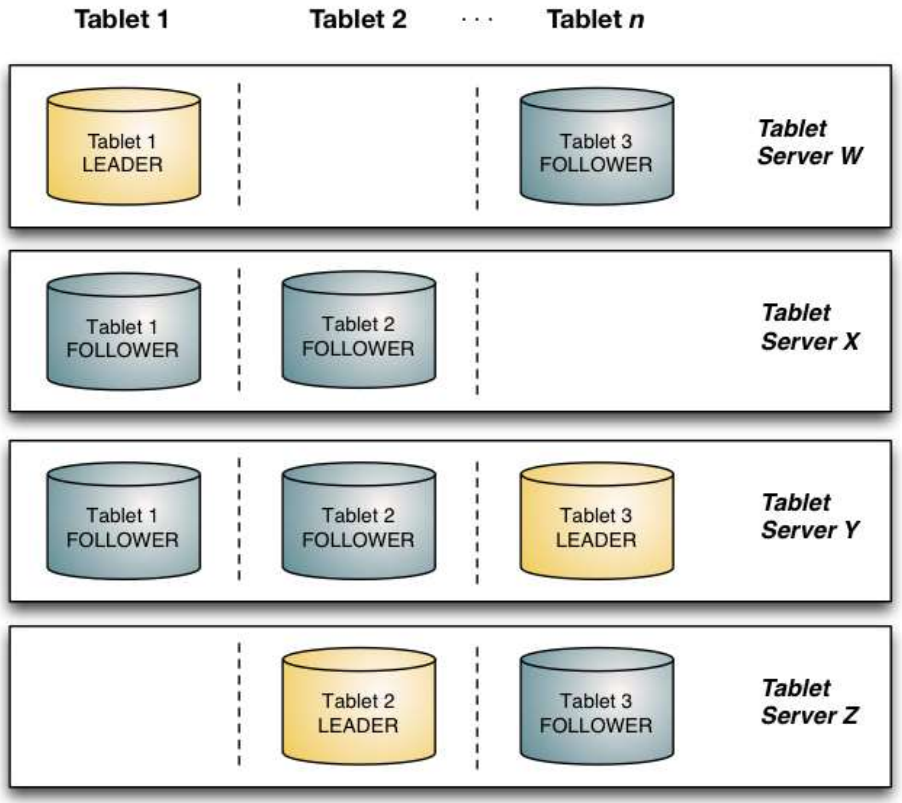
Meta Cache
T1: …
T2: …
T3: …

Client

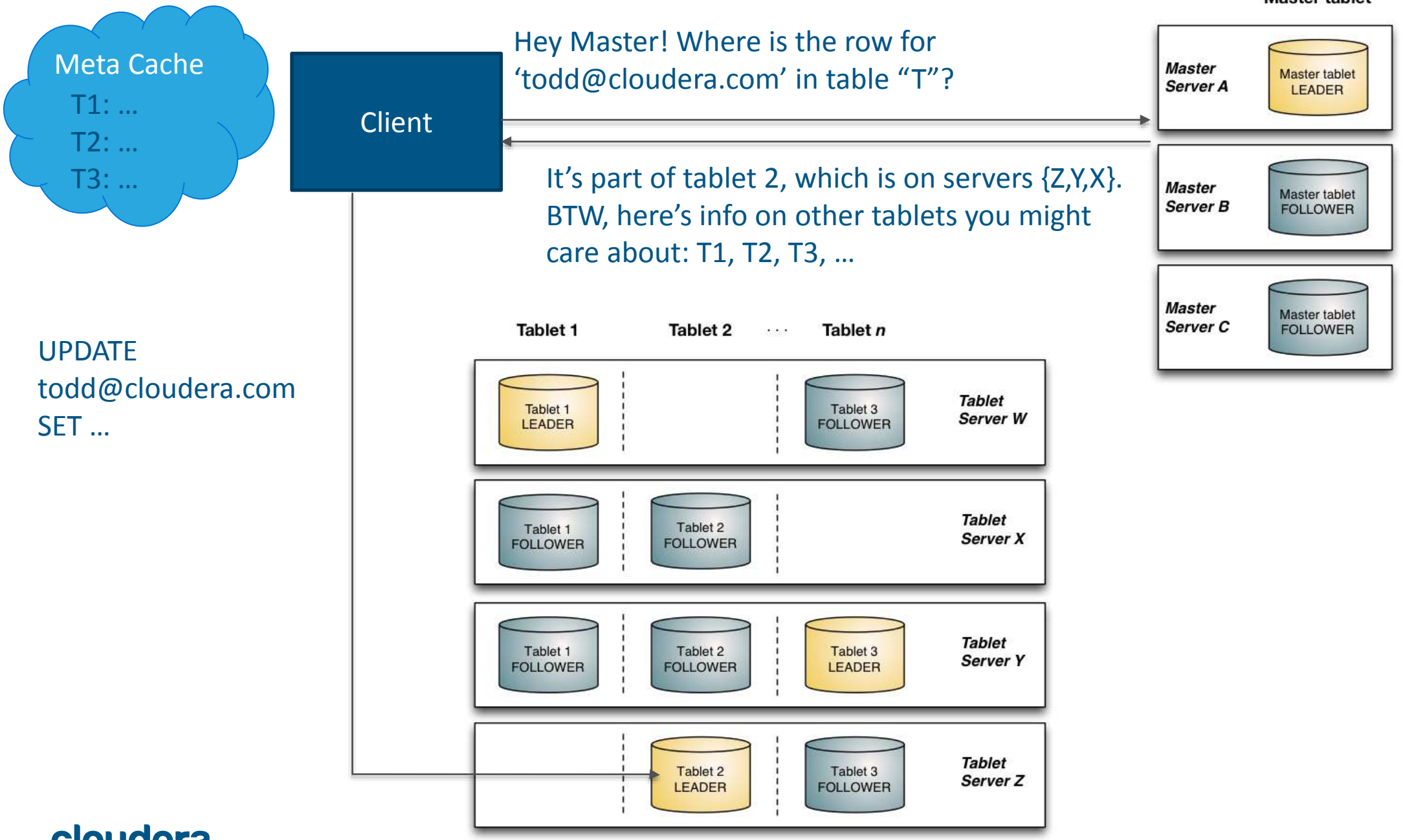
Hey Master! Where is the row for 'todd@cloudera.com' in table "T"?

It's part of tablet 2, which is on servers {Z,Y,X}. BTW, here's info on other tablets you might care about: T1, T2, T3, …





cloudera



UPDATE
todd@cloudera.com
SET …

cloudera

Tablet Design

- Inserts buffered in an in-memory store (like HBase's memstore)
- Flushed to disk
  - Columnar layout, similar to Apache Parquet
- Updates use MVCC (updates tagged with timestamp, not *in-place*)
  - Allow "SELECT AS OF <timestamp>" queries and consistent cross-tablet scans
- Near-optimal read path for "current time" scans
  - No per row branches, fast vectorized decoding and predicate evaluation
- Performance worsens based on number of recent updates

Metadata

- **Replicated master***
  - Acts as a tablet directory (“META” table)
  - Acts as a catalog (table schemas, etc)
  - Acts as a load balancer (tracks TS liveness, re-replicates under-replicated tablets)
- **Caches all metadata in RAM** for high performance
  - 80-node load test, GetTableLocations RPC perf:
    - 99th percentile: 68us, 99.99th percentile: 657us
    - <2% peak CPU usage
- Client configured with master addresses
  - Asks master for tablet locations as needed and caches them

Kudu Trade-Offs

- Random updates will be slower
  - HBase model allows random updates without incurring a disk seek
  - Kudu requires a key lookup before update, Bloom lookup before insert

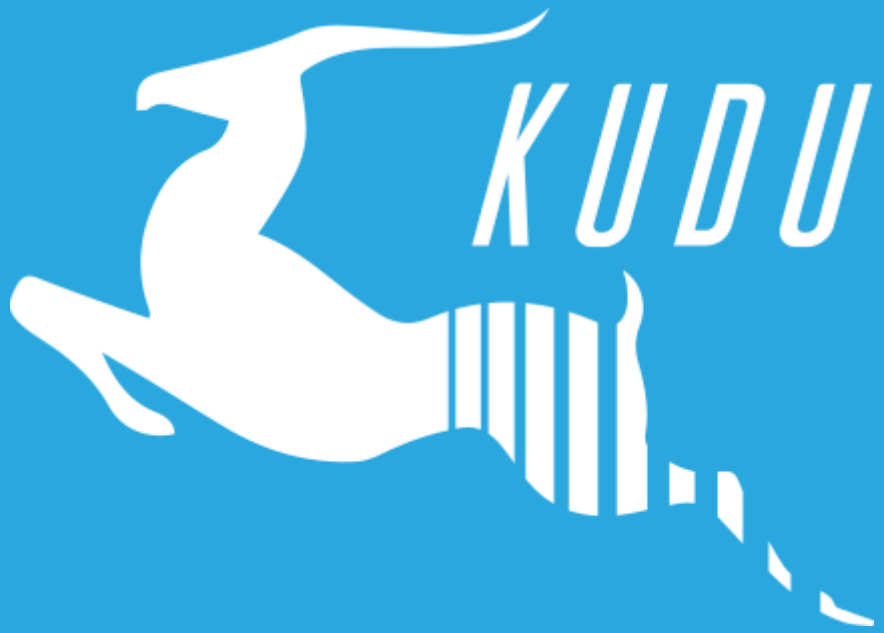
- Single-row reads may be slower
  - Columnar design is optimized for scans
  - Future: may introduce “column groups” for applications where single-row access is more important

Fault tolerance

- Transient FOLLOWER failure:
  - Leader can still achieve majority
  - Restart follower TS within 5 min and it will rejoin transparently
- Transient LEADER failure:
  - Followers expect to hear a heartbeat from their leader every 1.5 seconds
  - 3 missed heartbeats: leader election!
    - New LEADER is elected from remaining nodes within a few seconds
  - Restart within 5 min and it rejoins as a FOLLOWER
- N replicas handle (N-1)/2 failures

Fault tolerance (2)

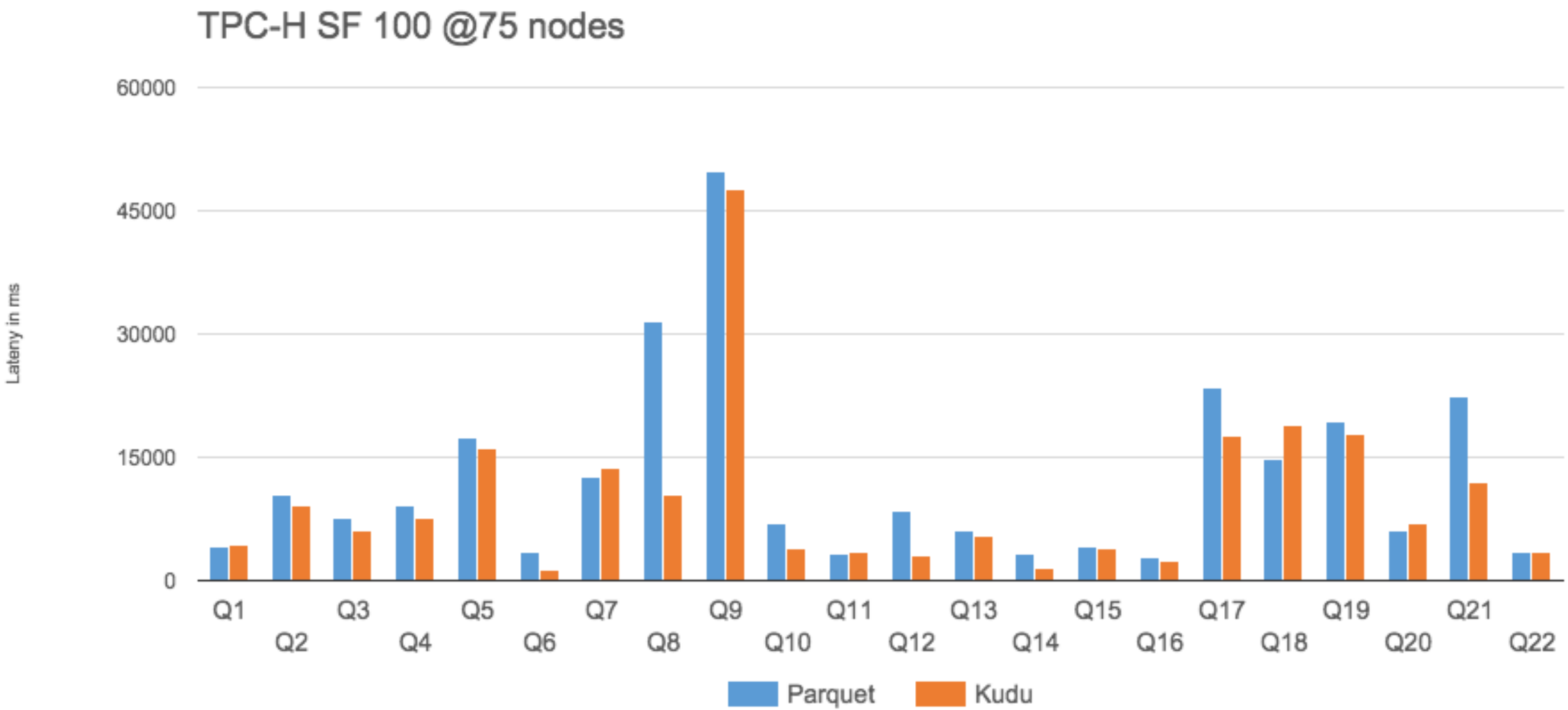
- Permanent failure:
  - Leader notices that a follower has been dead for 5 minutes
  - Evicts that follower
  - Master selects a new replica
  - Leader copies the data over to the new one, which joins as a new FOLLOWER



Benchmarks

TPC-H (Analytics benchmark)

- 75TS + 1 master cluster
  - 12 (spinning) disk each, enough RAM to fit dataset
  - Using Kudu 0.5.0, Impala 2.2 with Kudu support, CDH 5.4
  - TPC-H Scale Factor 100 (100GB)
- Example query:
  - ```
    SELECT n_name, sum(l_extendedprice * (1 - l_discount)) as revenue FROM customer,
    orders, lineitem, supplier, nation, region WHERE c_custkey = o_custkey AND
    l_orderkey = o_orderkey AND l_suppkey = s_suppkey AND c_nationkey = s_nationkey
    AND s_nationkey = n_nationkey AND n_regionkey = r_regionkey AND r_name = 'ASIA'
    AND o_orderdate >= date '1994-01-01' AND o_orderdate < '1995-01-01’ GROUP BY
    n_name ORDER BY revenue desc;
    ```
    ```

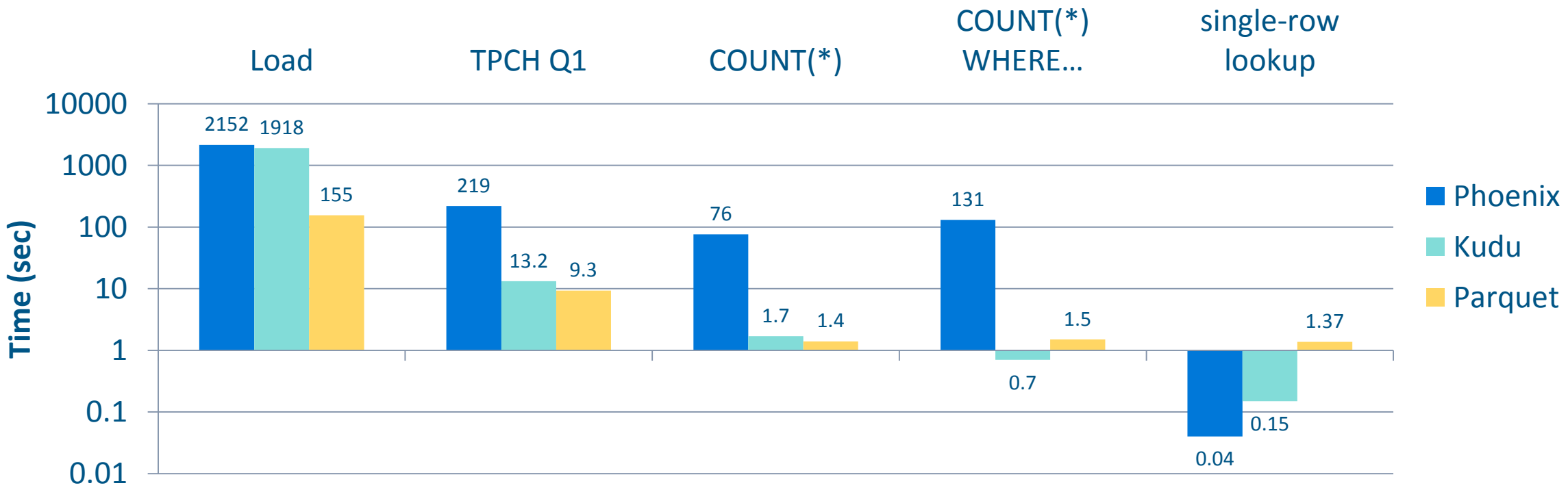


- Kudu outperforms Parquet by 31% (geometric mean) for RAM-resident data
- Parquet likely to outperform Kudu for HDD-resident (larger IO requests)

cloudera

What about Apache Phoenix?

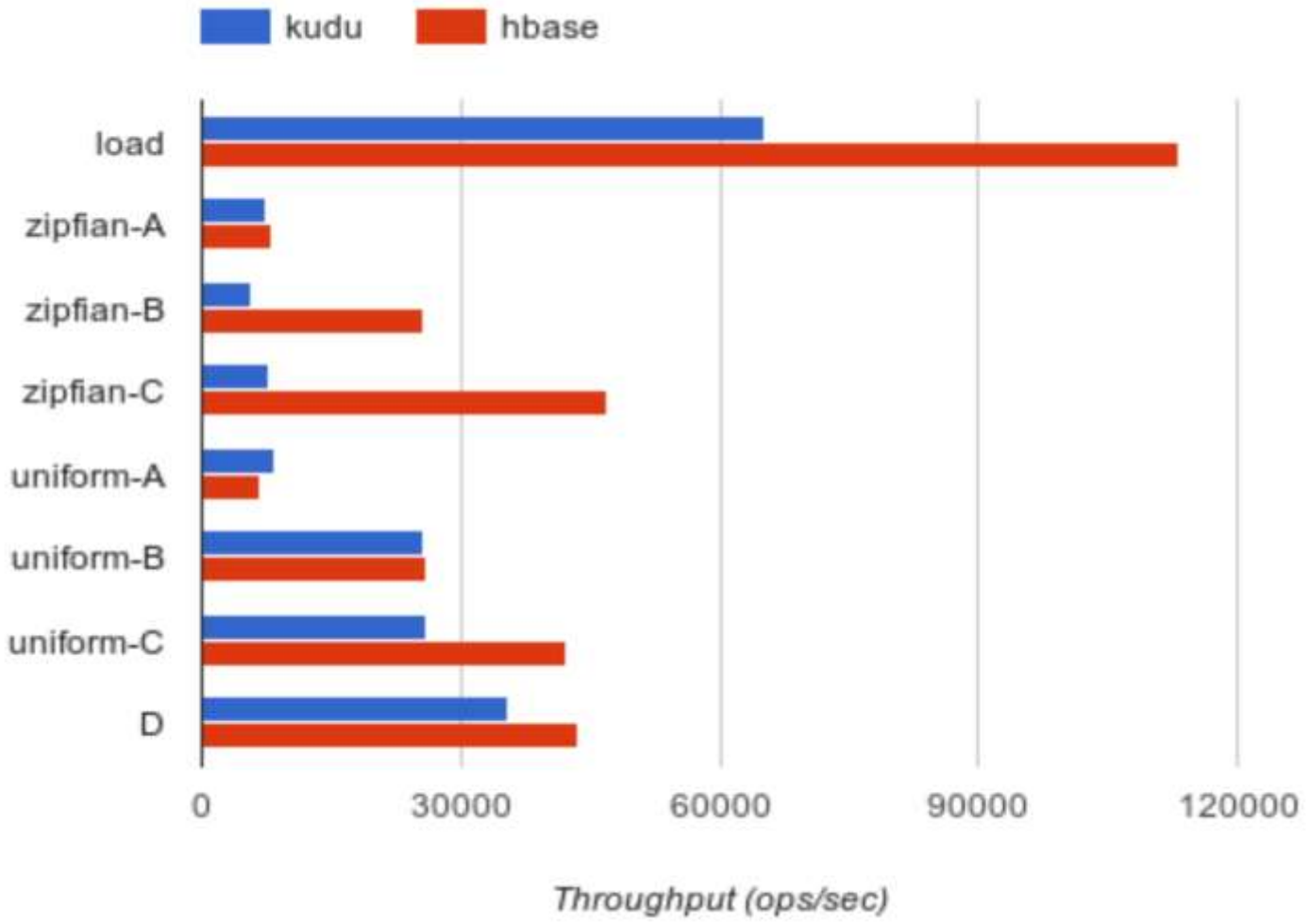
- 10 node cluster (9 worker, 1 master)
- HBase 1.0, Phoenix 4.3
- TPC-H LINEITEM table only (6B rows)



cloudera

What about NoSQL-style random access? (YCSB)

- **YCSB** 0.5.0-snapshot
- 10 node cluster
  (9 worker, 1 master)
- HBase 1.0
- 100M rows, 10M ops



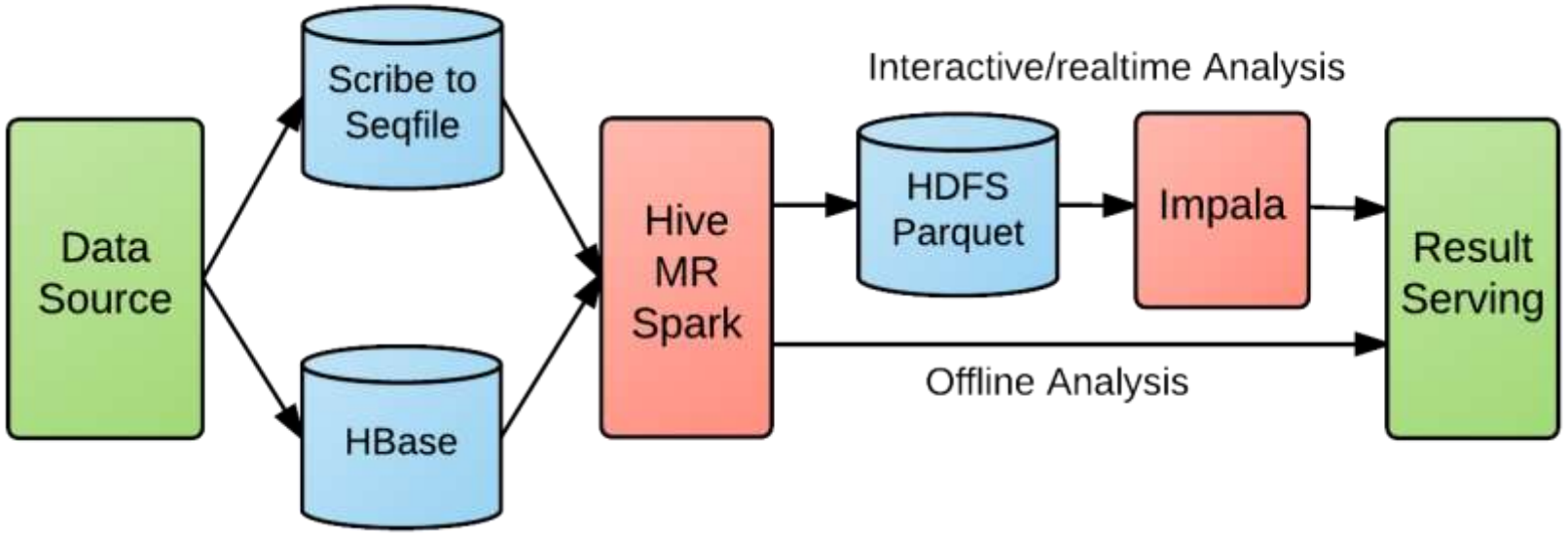
Xiaomi Use Case

- Gather important RPC tracing events from mobile app and backend service
- Service monitoring & troubleshooting tool
- **High write throughput**
  - >5 Billion records/day and growing
- **Query latest data and quick response**
  - Identify and resolve issues quickly
- **Can search for individual records**
  - Easy for troubleshooting

cloudera

Big Data Analytics Pipeline

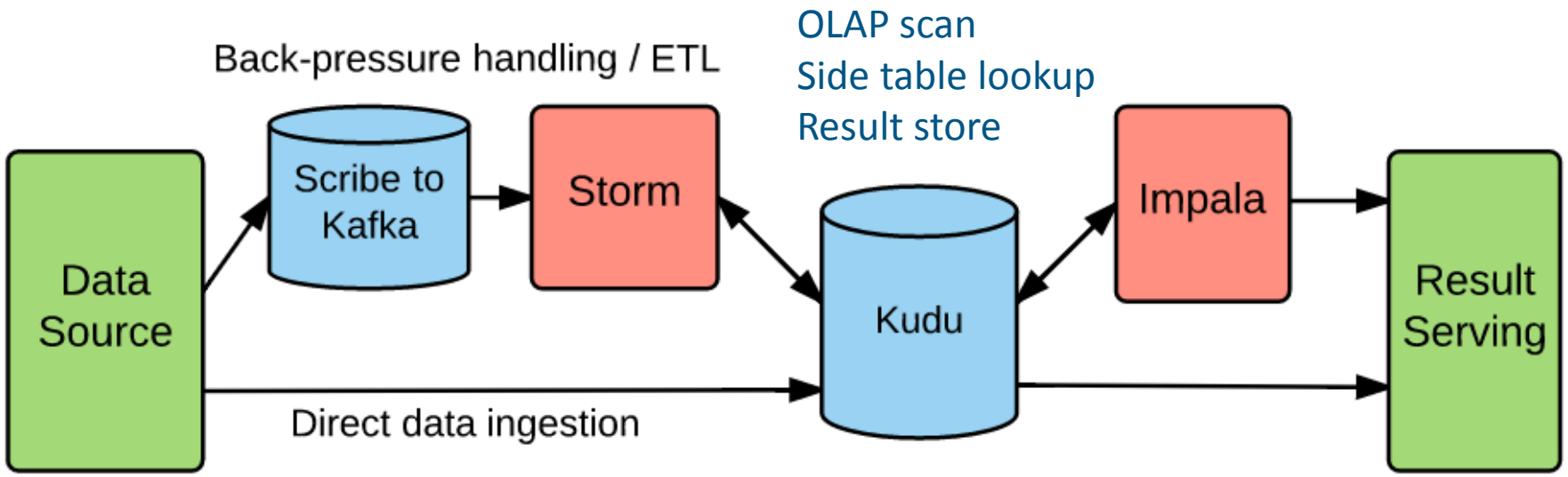
Before Kudu



- **Long pipeline**
  high latency(1 hour ~ 1 day), data conversion pains
- **No ordering**
  Log arrival(storage) order not exactly logical order
  e.g. read 2-3 days of log for data in 1 day

Big Data Analysis Pipeline

Simplified With Kudu



- ETL Pipeline(0~10s latency)
  Apps that need to prevent backpressure or require ETL
- Direct Pipeline(no latency)
  Apps that don’t require ETL and no backpressure issues

Use Case 1: Benchmark

Environment

- 71 Node cluster
- Hardware
  - CPU: E5-2620 2.1GHz * 24 core Memory: 64GB
  - Network: 1Gb Disk: 12 HDD
- Software
  - Hadoop2.6/Impala 2.1/Kudu

Data

- 1 day of server side tracing data
  - ~2.6 Billion rows
  - ~270 bytes/row
  - 17 columns, 5 key columns

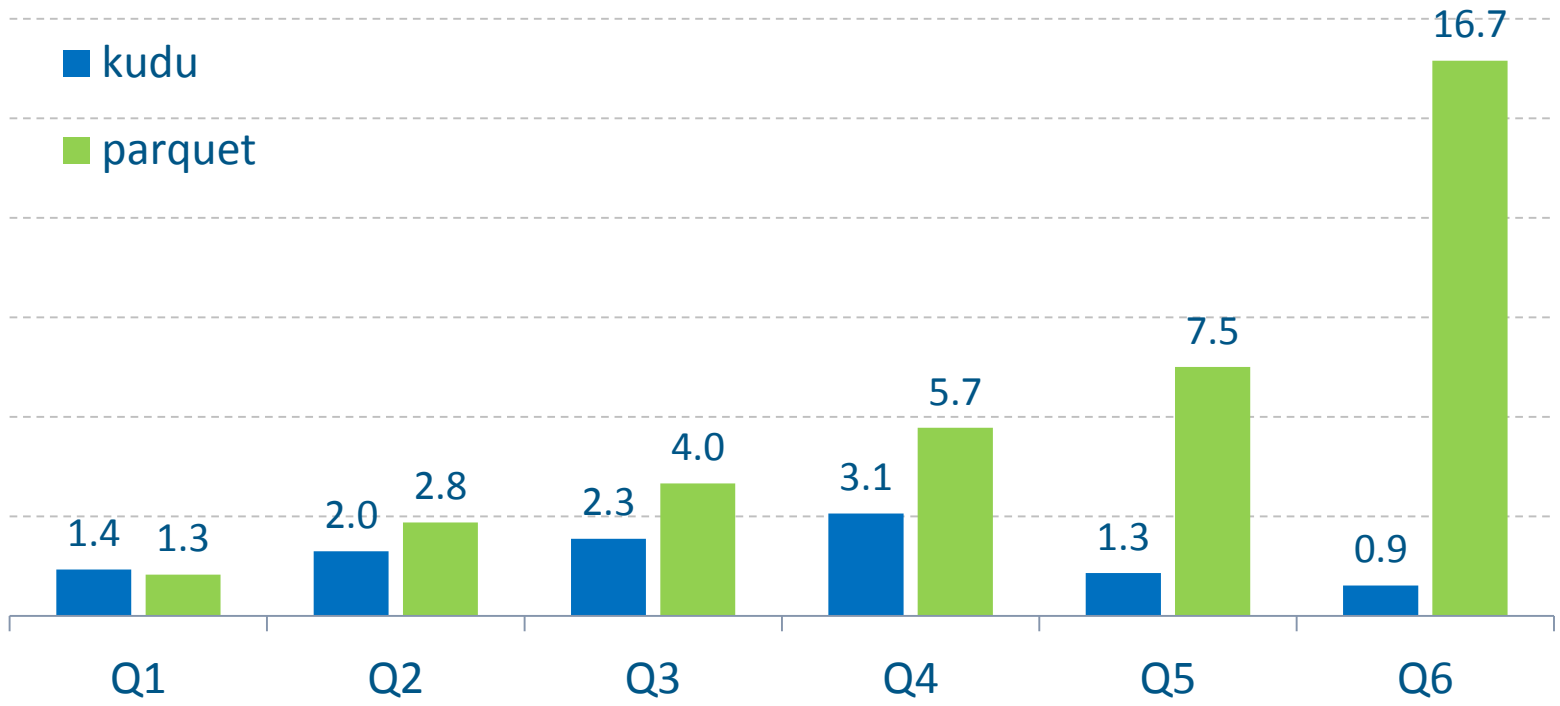
cloudera

Use Case 1: Benchmark Results

Bulk load using impala (INSERT INTO):

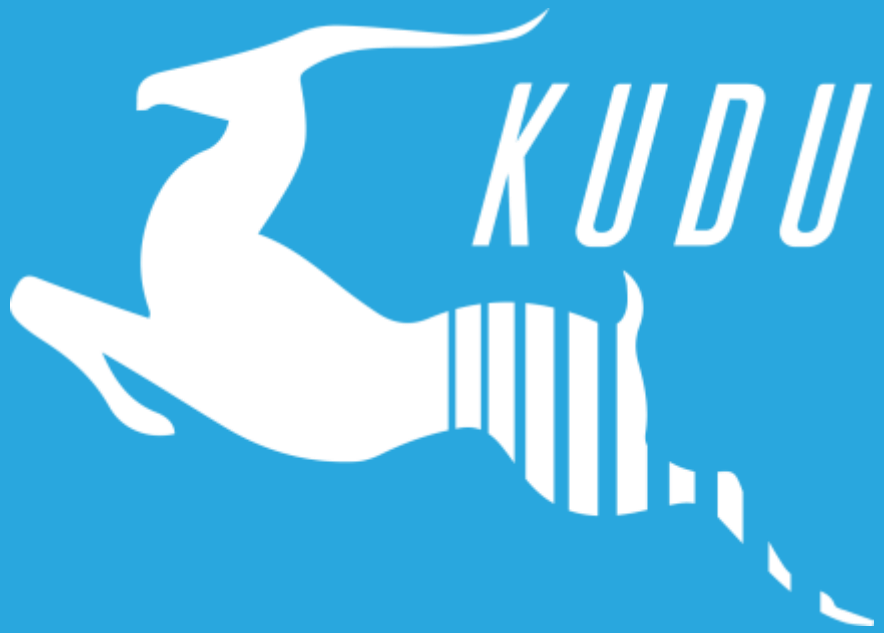
	Total Time(s)	Throughput(Total)	Throughput(per node)
Kudu	961.1	2.8M record/s	39.5k record/s
Parquet	114.6	23.5M record/s	331k records/s

Query latency:



* HDFS parquet file replication = 3 , kudu table replication = 3
* Each query run 5 times then take average

cloudera



Status & Getting Started

cloudera

Basic Kudu value proposition

With Kudu:

- Ingest and serve data simultaneously
- Support analytic and real-time operations on the same data set
- Make existing storage architectures simpler, and enable new architectures that previously weren't possible

Basic Features:

- High availability, no single point of failure
- Consistency by consensus, options for "tunable consistency"
- Horizontally scalable
- Efficient use of modern storage and processors

cloudera

Current Status

- ✓ Completed all components core to the architecture
- ✓ Java and C++ API
- ✓ Impala, MapReduce, and Spark integration
- ✓ Support for SSDs and spinning disk
- ✓ Fault recovery
- ✓ Public beta available

cloudera

Getting Started

Users:

Install the Beta or try a VM:
getkudu.io

Get help:
kudu-user@googlegroups.com

Read the white paper:
getkudu.io/kudu.pdf

Developers:

Contribute:
github.com/cloudera/kudu (commits)
gerrit.cloudera.org (reviews)
issues.cloudera.org (JIRAs going back to 2013)

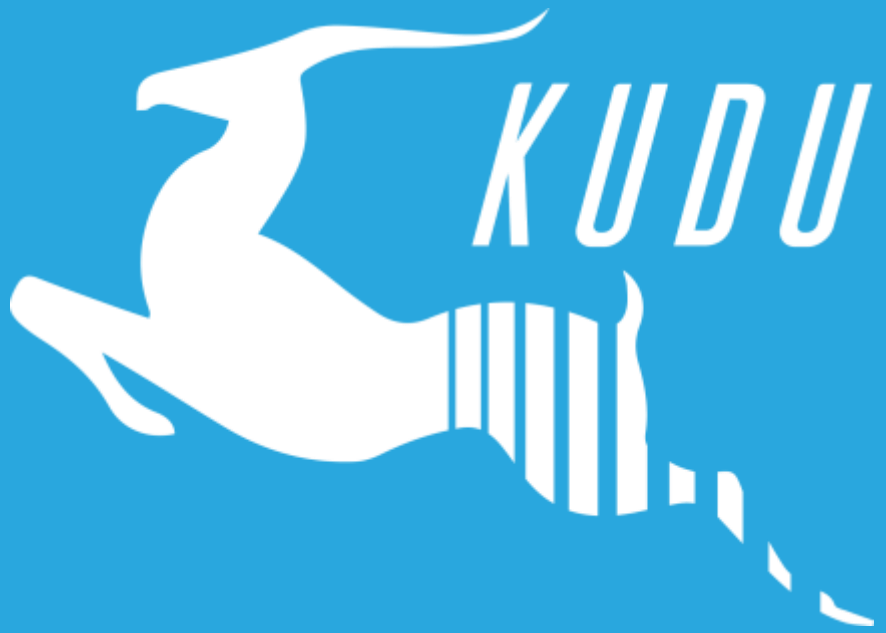
Join the Dev list:
kudu-dev@googlegroups.com

Contributions/participation are welcome and encouraged!

cloudera

cloudera

Questions?



Appendix

cloudera

Fault tolerance

- Transient FOLLOWER failure:
  - Leader can still achieve majority
  - Restart follower TS within 5 min and it will rejoin transparently
- Transient LEADER failure:
  - Followers expect to hear a heartbeat from their leader every 1.5 seconds
  - 3 missed heartbeats: leader election!
    - New LEADER is elected from remaining nodes within a few seconds
  - Restart within 5 min and it rejoins as a FOLLOWER
- N replicas handle (N-1)/2 failures

Fault tolerance (2)

- Permanent failure:
  - Leader notices that a follower has been dead for 5 minutes
  - Evicts that follower
  - Master selects a new replica
  - Leader copies the data over to the new one, which joins as a new FOLLOWER

LSM vs Kudu

- LSM – Log Structured Merge (Cassandra, HBase, etc)
  - Inserts and updates all go to an in-memory map (MemStore) and later flush to on-disk files (HFile/SSTable)
  - Reads perform an on-the-fly merge of all on-disk HFiles
- Kudu
  - Shares some traits (memstores, compactions)
  - More complex.
  - **Slower writes** in exchange for **faster reads (especially scans)**

Kudu storage – Compaction policy

- Solves an optimization problem (knapsack problem)
- Minimize “height” of rowsets for the average key lookup
  - Bound on number of seeks for write or random-read
- Restrict total IO of any compaction to a budget (128MB)
  - **No long compactions, ever**
  - **No “minor” vs “major” distinction**
  - Always be compacting or flushing
  - Low IO priority maintenance threads