From Zero to Hero with Apache Kudu

BOSS 2019

Andrew Wong

Follow along with the walkthroughs!

https://bit.ly/2ZsR6m6

Who am I?

Andrew Wong

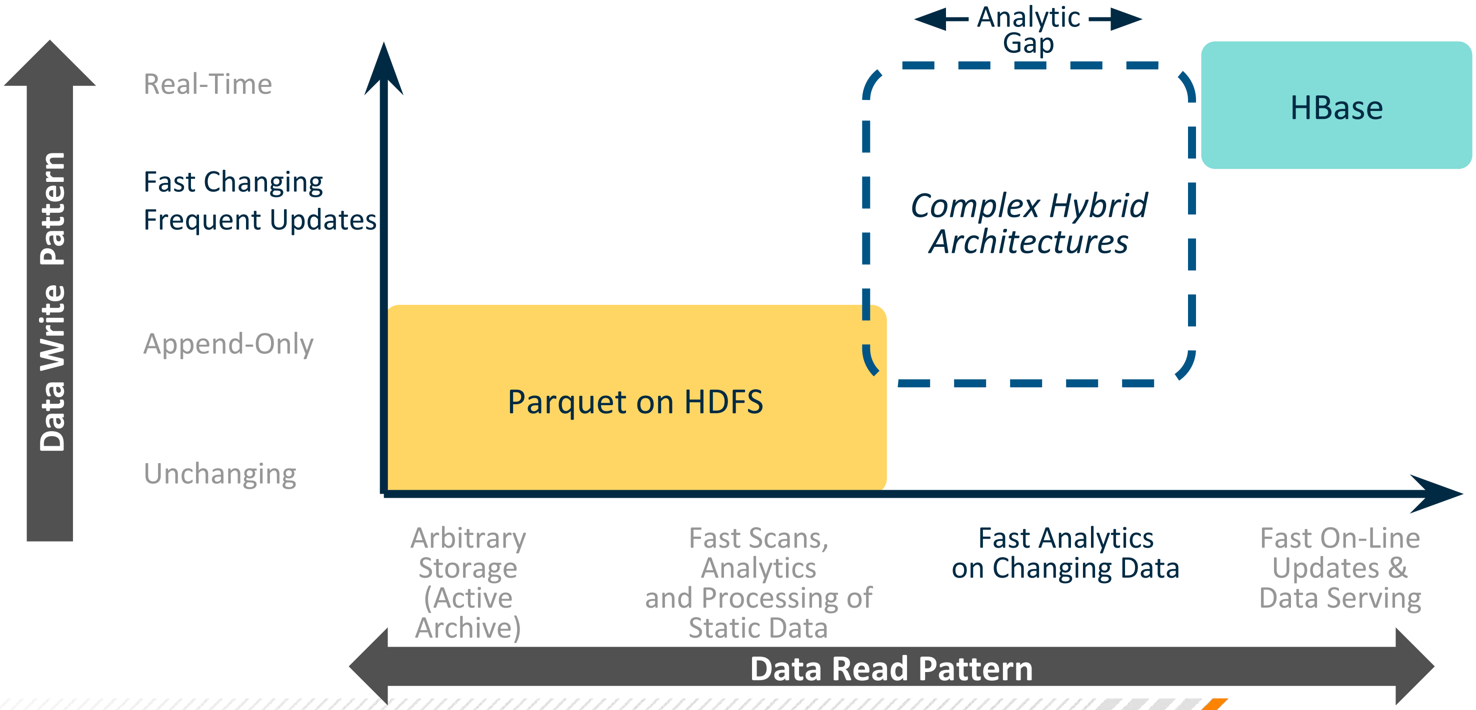
- Software engineer at Cloudera (awong@cloudera.com)
- Apache Kudu PMC Member

Some Kudu things I have worked on:

- Scan optimizations
- Disk failure mitigation
- Integration with Hive Metastore (external catalog)
- Fine-grained authorization

Traditional big data storage leaves a gap

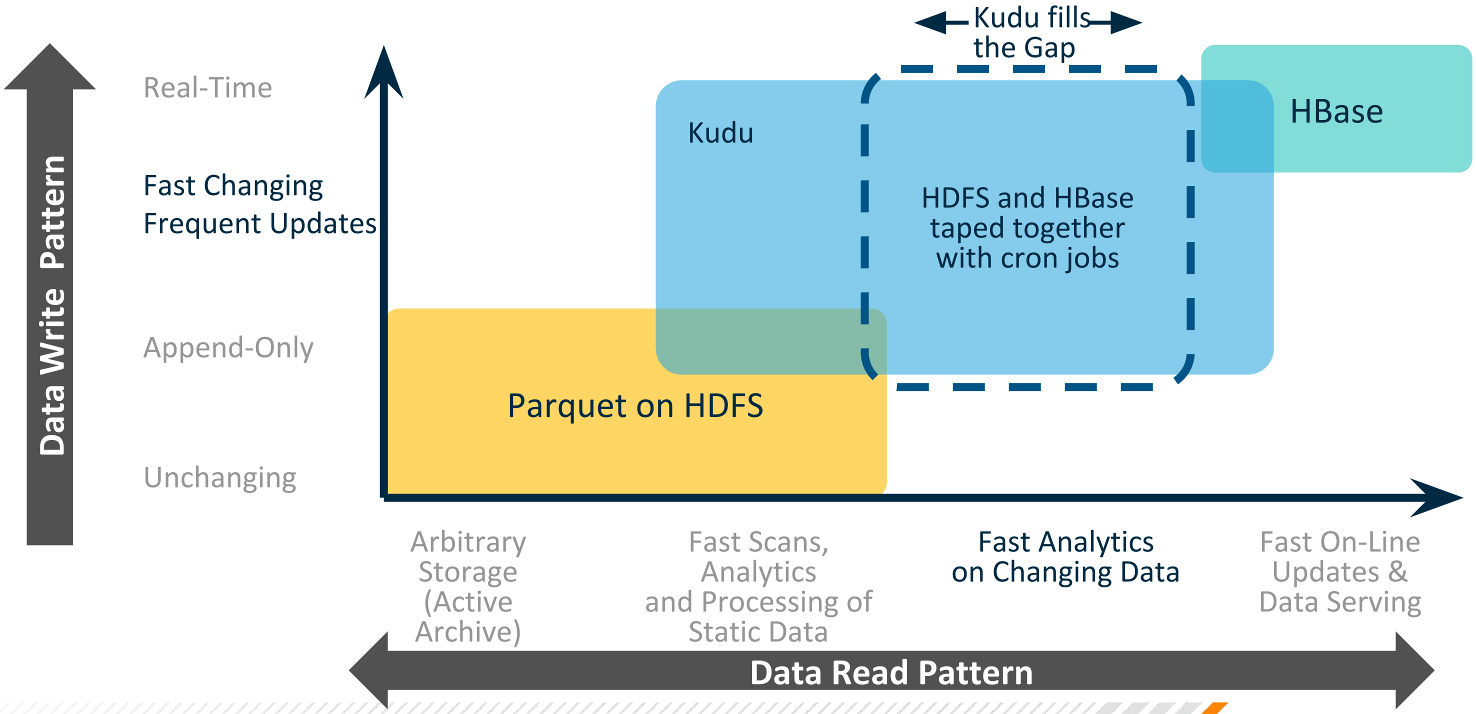
Use cases fall between HDFS and HBase were difficult to manage



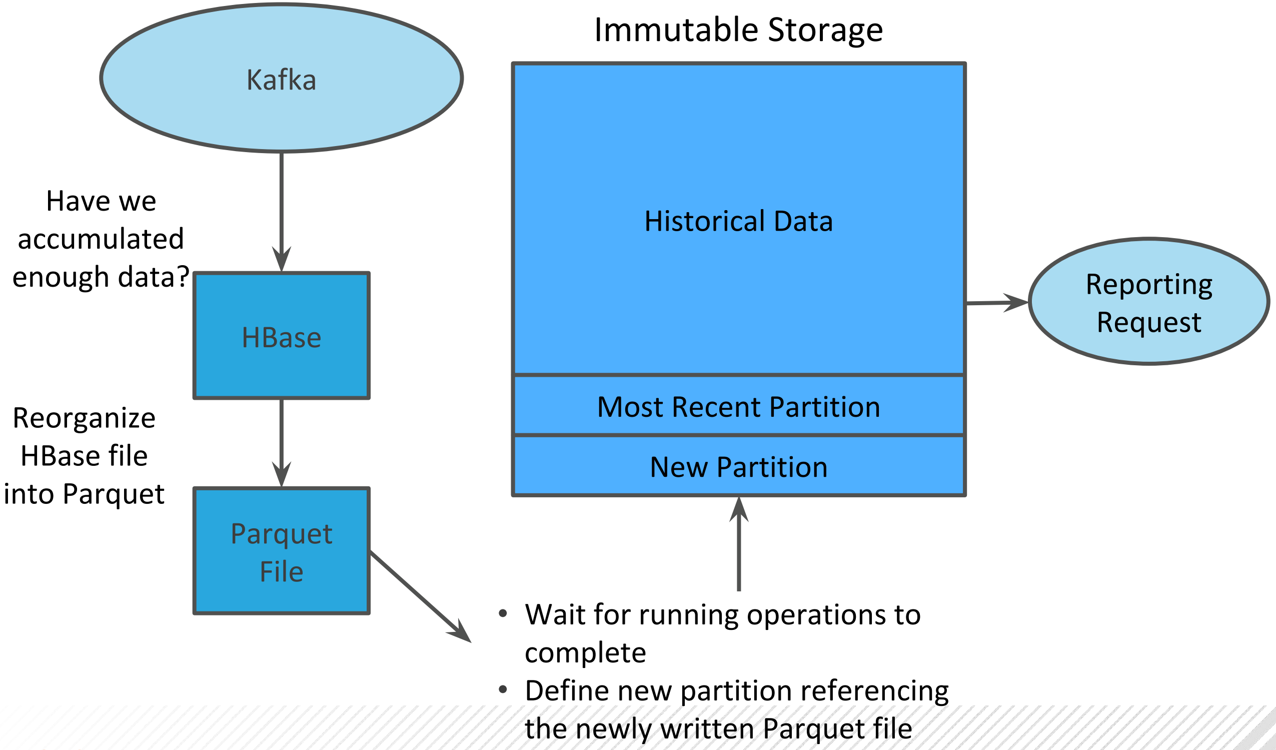
CLOUDERA

Traditional big data storage leaves a gap

Use cases fall between HDFS and HBase were difficult to manage



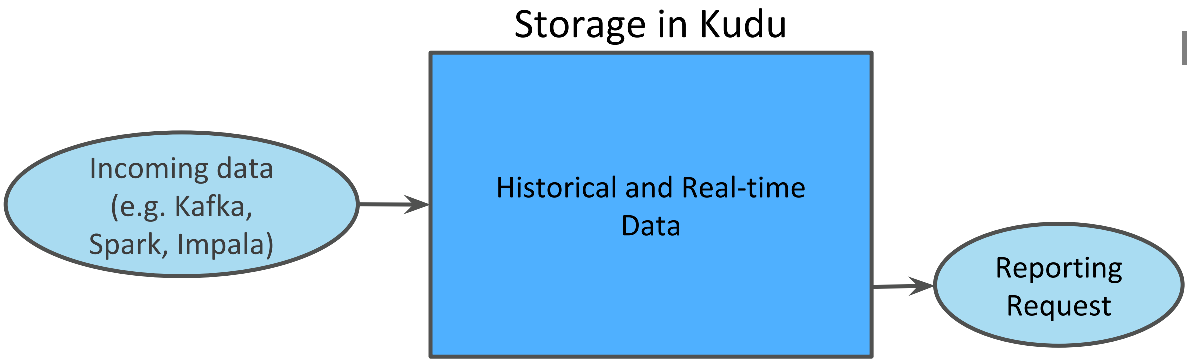
“Traditional” real-time analytics



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?

Real-time analytics with Kudu



Improvements:

- Much simpler architecture
- Significantly easier to handle late arrivals of data
- New data available immediately for analytics or operations

What is Apache Kudu?

Mutable data storage engine, designed for analytics on real-time data

MUTABLE

STORAGE

ANALYTICS

REAL-TIME DATA

What is Apache Kudu?

Mutable data storage engine, designed for analytics on real-time data

MUTABLE

STORAGE

ANALYTICS

REAL-TIME DATA

What is Apache Kudu?

Mutable data storage engine, designed for analytics on real-time data

MUTABLE

STORAGE

ANALYTICS

REAL-TIME DATA

What is Apache Kudu?

Mutable data storage engine, designed for analytics on real-time data

MUTABLE

STORAGE

ANALYTICS

REAL-TIME DATA

What is Apache Kudu?

Mutable data storage engine, designed for analytics on real-time data

MUTABLE

STORAGE

ANALYTICS

REAL-TIME DATA

“Big deal... Aren’t there a ton of big data systems out there?”

Yes there are.

- Open source
  - Parquet on HDFS, object storage
  - HBase/Cassandra
  - TiDB with TiFlash
  - etc.
- Proprietary
  - Vertica, Teradata, SAP
  - Spanner
  - Redshift

Why else did we build Kudu?

Changing hardware landscape

- Spinning disks --> solid state storage
  - **NAND flash:** up to 450k read, 250k write IOPS, ~2GB/s read and ~1.5GB/s write throughput, at under $1/GB
  - **PMEM:** order of magnitude faster than NAND, cheaper than RAM
- **RAM** is getting cheaper and more abundant
  - 128 --> 256 --> 512 GB over the last few years

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

Scalable and fast tabular storage

Scalable

- Production clusters with hundreds of nodes
- On the order of hundreds of TBs to low PBs

Fast

- Written primarily in C++
- Millions of write operations per second across cluster
- Multiple GB/s read throughput per node

Tabular

- Strict schema, finite column count, no BLOBs

And we've got some friends!

Apache and CNCF

And we've got some friends!

Apache and CNCF

New in 2018/19!

In progress (Bahir)

Most commonly used

Dev only

The rest of the Ecosystem isn't that scary

Somewhat clear lines between roles

Ingest:

- NiFi, Spark Streaming, Flink, Kafka, Flume

Storage:

- **Kudu**
- Previously HBase, HDFS, and a lot of cron jobs

Querying:

- Impala, Hive, Spark SQL

From Zero to Hero with Apache Kudu

Agenda

Start with the basics:

- Introduction to Kudu's data model
- Distributed architecture

Jump into more complex things:

- Schema design exercises
- Partitioning in Kudu

Deploy a small "big data" pipeline:

- Spark, Nifi

From Zero to Hero with Apache Kudu

If you can

- Pull from GitHub:
  - `apache/kudu:master`
- Pull from DockerHub:
  - `apache/kudu:latest`
- `brew install apache-spark`

A primer on Kudu concepts

What is “data” to Kudu?

Table: prominent abstraction for users -- a set of uniquely identifiable rows

Schema: describes the columns and ordering of the rows of a table

- Subset of columns defined as “primary key”

Partition schema: describes the partitions of a table

- Subset of primary key defined as “partition key”

Tablet: partition of a table; the logical unit of replication and parallelization

- Underlying data stored in sorted order by primary key
- Tablet replicas are dispersed among servers in a cluster

What makes a tablet?

Very much LSM-inspired

Write-ahead log: as writes come in, they are written in fast, row-oriented storage

Mem-Rowset: as writes are written to the WAL, they are also applied to an in-memory, row-oriented B-Tree

Periodic flushes: as the mem-rowset grows large, there are periodic flushes to free up memory and transition the mem-rowset to a disk-rowset

Many Disk-Rowsets: represented as an interval tree for lookups

What makes a tablet?

Wait what is “LSM”? You said we’re starting from zero!

- 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)

What makes a tablet?

And more!

B-Tree index per rowset to enable search on a primary key

Bloom filter per rowset to enable lookups for deduplication of rows

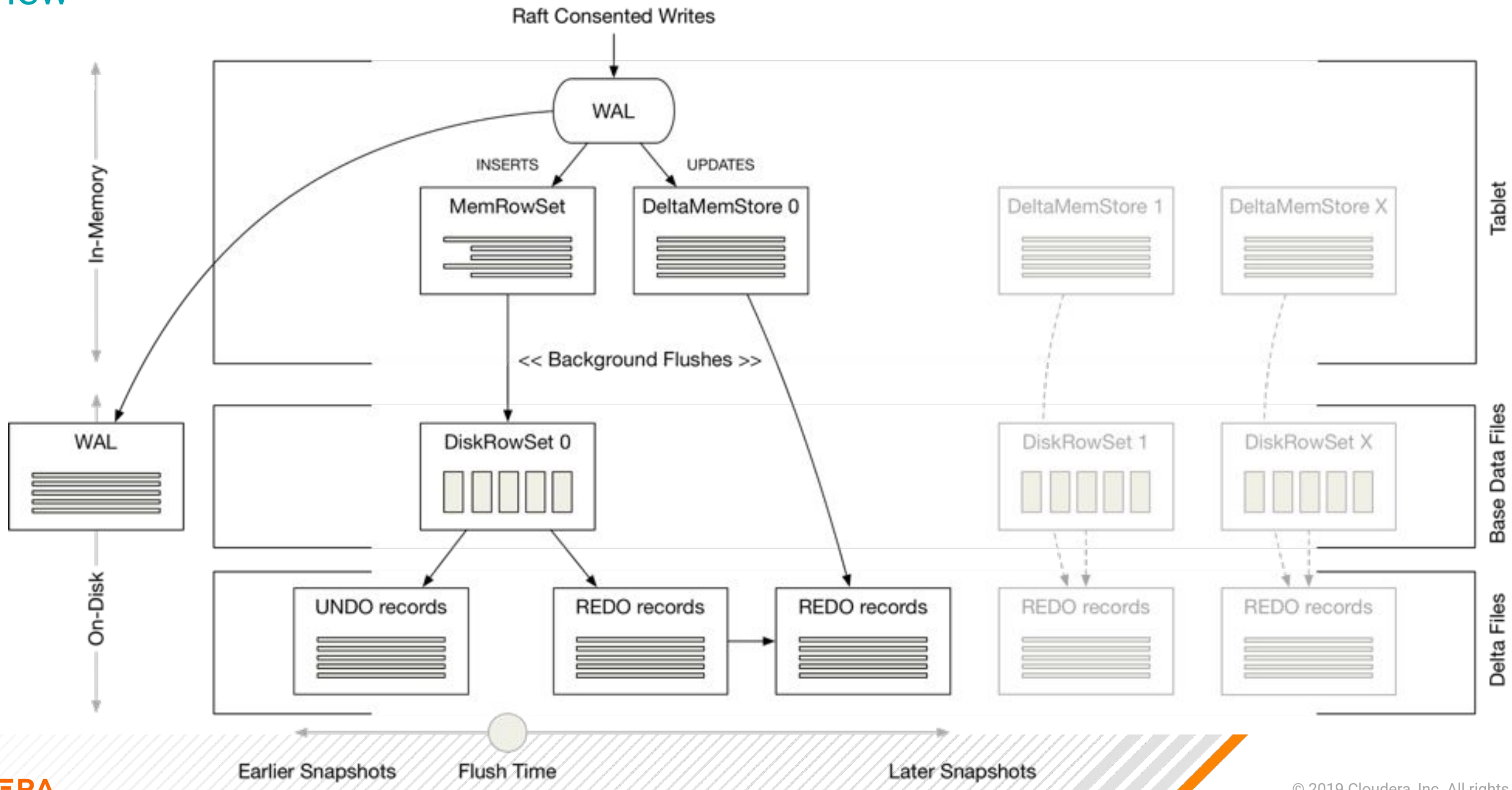
CFiles the actual columnar data, in primary key order

Deltas (updates and deletes)

- **In-memory delta store** per rowset
- **On-disk delta stores** per rowset
- New updates are represented as REDO records, periodically flushed to UNDO records to avoid REDO traversal

What makes a tablet?

Overview



CLOUDERA

Maintenance Manager

Always be flushing and compacting!

- The maintenance manager handles background tasks
  - Flushing the MRS or DMS to disk
  - Compactions
  - WAL GC
- A maintenance manager thread decides what task to perform using a cost-based optimization model
  - Prefers flushing when the server is under memory pressure

Distributed architecture

Main roles in Kudu

Master: a server that hosts Kudu system metadata and catalog information

Tablet server: a server that hosts tablet replicas

Client: an application that inserts to or reads from Kudu

- Tooling
- C++ application
- Java
- Python
- Spark
- Impala
- ...

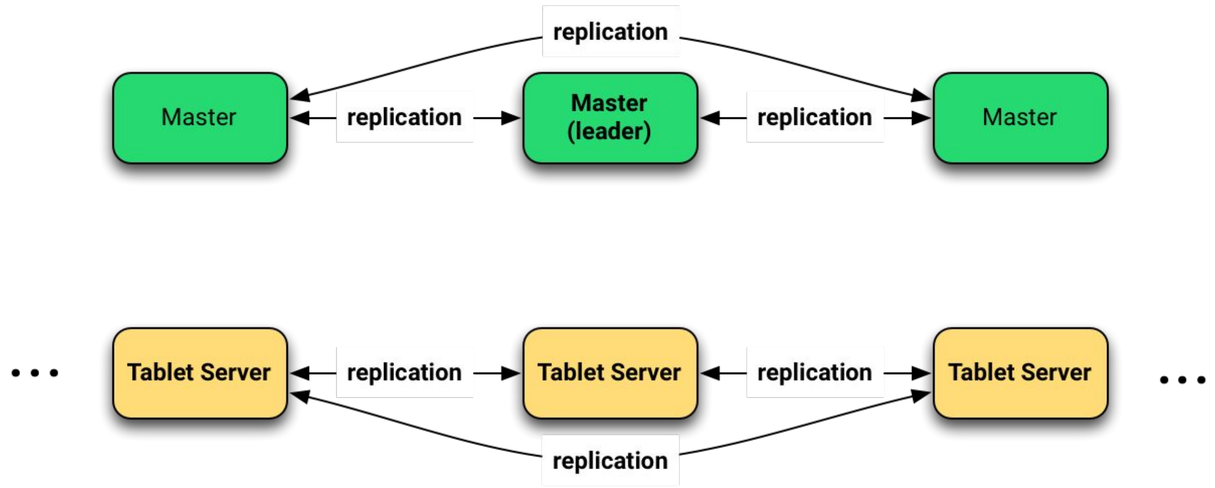
Let’s walk through deploying Kudu!

https://bit.ly/2ZsR6m6

- Explore a single node deployment of Kudu

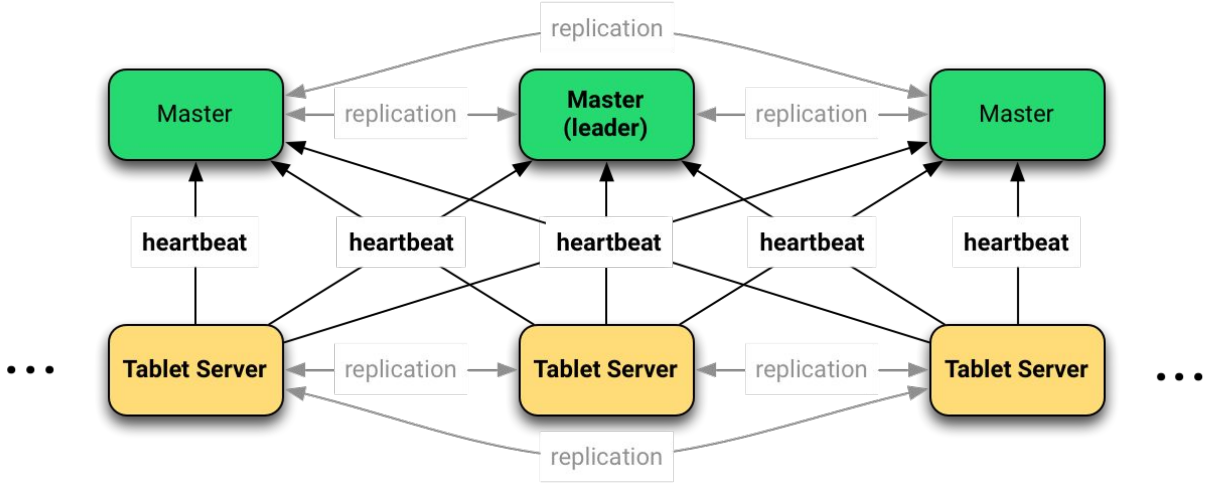
Distributed architecture

Replication, replication, replication!



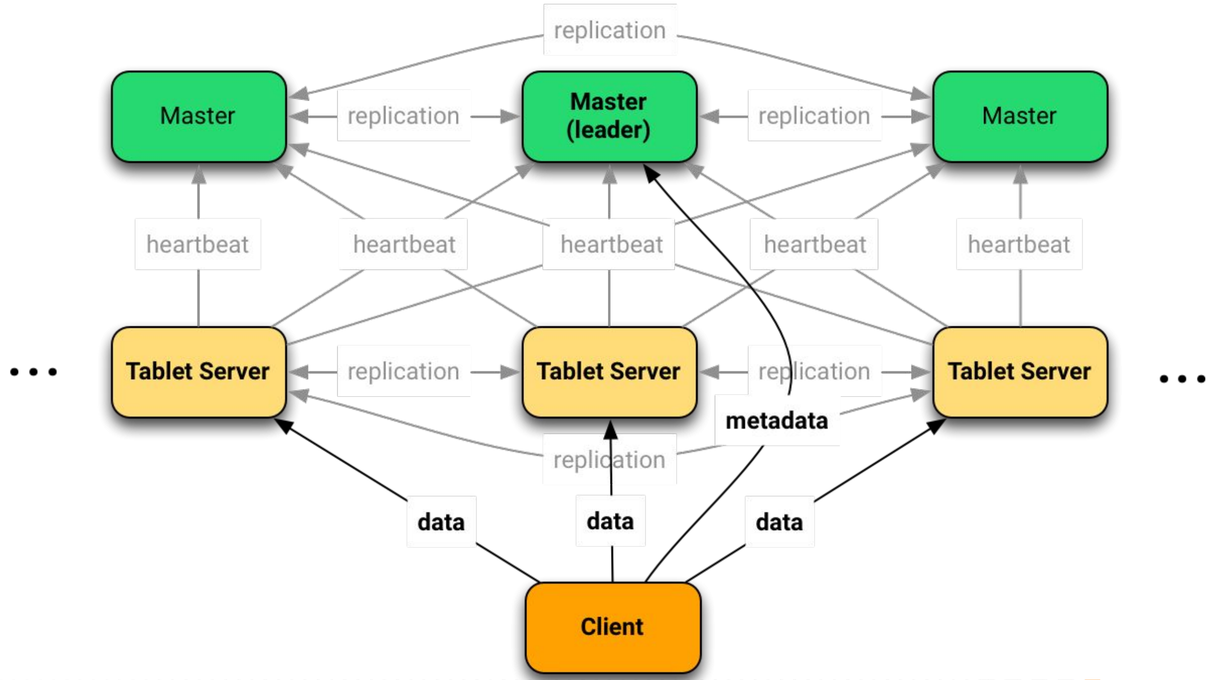
Distributed architecture

Replication, replication, replication!



Distributed architecture

Replication, replication, replication!

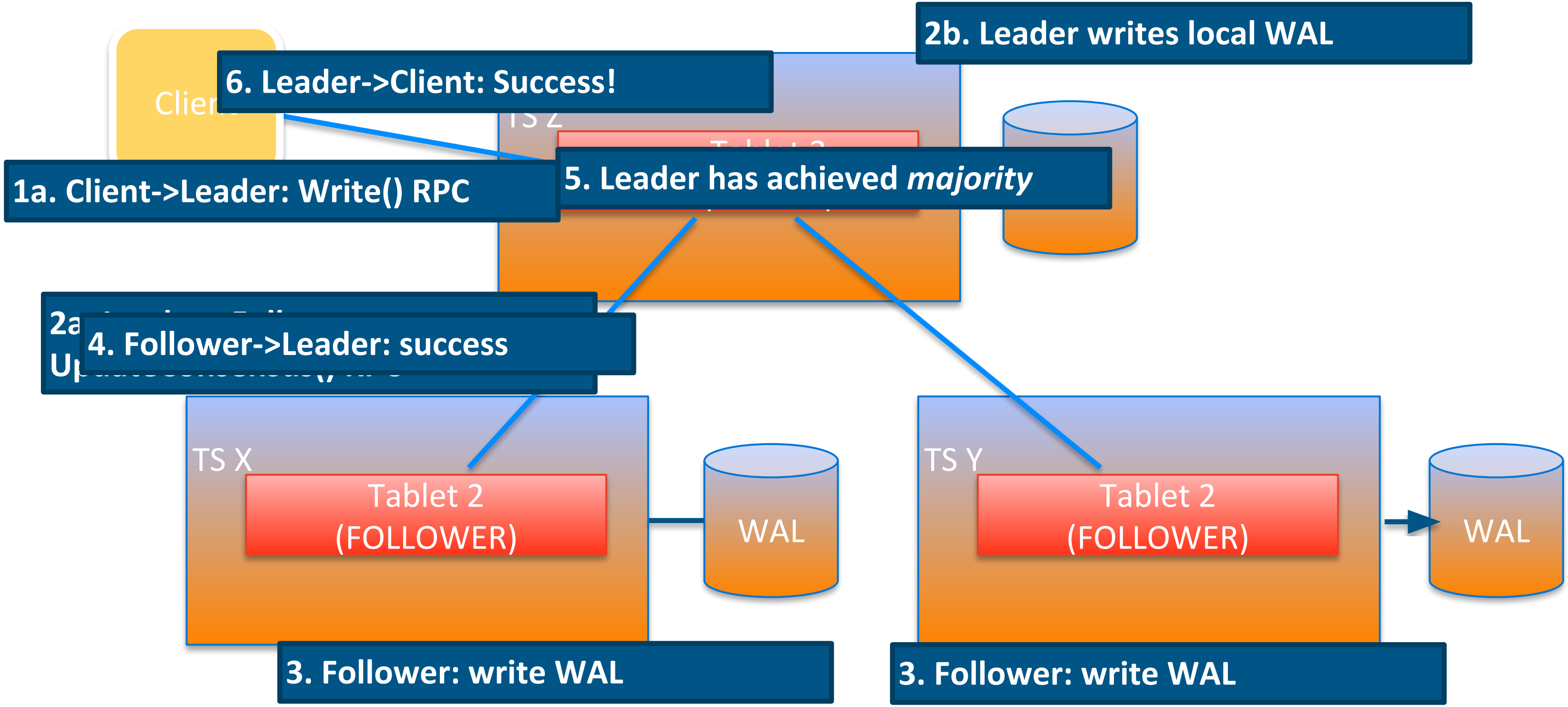


Raft consensus

Consensus protocol to replicate data

- Leaders constantly heartbeating to followers
- Leader elections triggered when a follower doesn't hear back from leader
- Write operations go to leader first and are replicated to followers

Raft consensus



Dealing with node failures

Wait long enough, and Kudu heals itself!

- If a Raft leader notices that it hasn't heard from one of its replicas in a while, it will try to create a new replica
- Physical copy of leader sent to new replica
- Old replica is removed from Raft configuration

Master details

- All table and tablet metadata is stored in a single tablet
  - Special tablet ID: 00000000000000000000000000000000
- Like any other tablet, the metadata tablet is replicated via Raft
- Master is effectively a tablet server for this one tablet
- Failure handling
  - Transient failures are handled transparently
  - Permanent failures require operator intervention

Balance and Skew

Tablet replica placement controlled by the master based on its view of the world

- Placement uses "Power of Two Choices" algorithm to even out replica count across cluster
  - Select two tablet servers, pick the one with fewer tablet replicas
- Rebalancer tool can be run to redistribute replicas in case of skew

CLOUDERA

Cluster health

kudu cluster ksck <masters>

When operating Kudu, this is your lifeline!

Tells you:

- What servers exist
- What tables and tablets exist
- What tablet are under-replicated and recovering
- What the skew on the cluster is like
- What non-default flags are set

Client abstractions

Sessions:

- In-memory write buffers, eventually get “flushed” to the servers

Scanners:

- Fetches rows one batch at a time

Client abstractions (writes)

```
KuduTable table = client.openTable(“metrics”);
KuduSession session = client.newSession();
Insert ins = table.newInsert();
ins.getRow().addString(“host”, “foo.example.com”);
ins.getRow().addString(“metric”, “load-avg.1sec”);
ins.getRow().addDouble(“value”, 0.05);
session.apply(ins);
session.flush();
```

Client abstractions (scans)

```
KuduScanner scanner = client.newScannerBuilder(table)
  .setProjectedColumnNames(Lists.of(“value”))
  .build();
while (scanner.hasMoreRows()) {
  RowResultIterator batch = scanner.nextRows();
  while (batch.hasNext()) {
    RowResult result = batch.next();
    System.out.println(result.getDouble(“value”));
  }
}
```

Client abstractions (scans, but with predicates!)

```
KuduScanner scanner = client.newScannerBuilder(table)
  .addPredicate(KuduPredicate.newComparisonPredicate(
    table.getSchema().getColumn("timestamp"),
    ComparisonOp.GREATER,
    System.currentTimeMillis() / 1000 + 60))
  .build();
```

Note: Kudu can evaluate simple predicates, but no aggregations, complex expressions, UDFs, etc.

Client cluster interaction

Master

- Tablet metadata fetched from the master, including locations and partitioning info
- Tablets are “pruned” via partitioning info so only the appropriate tablets are scanned

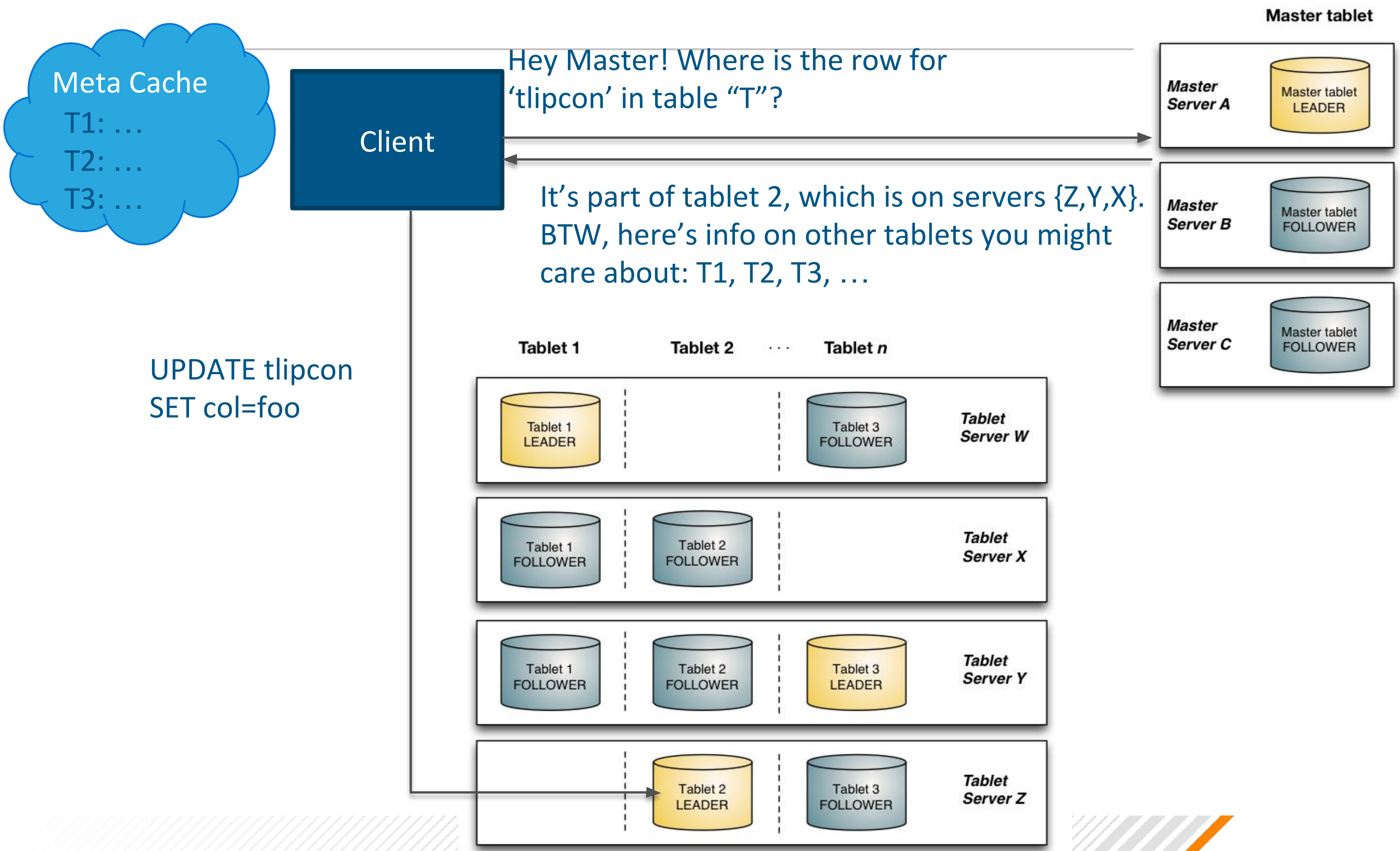
Tablet servers

- Requests sent to tablet servers
- Rejected if appropriate

Client consistency models

Choose the one which fits your workload!

- READ_LATEST (default mode)
  - Read committed state immediately
- READ_AT_SNAPSHOT
  - Consistent and repeatable
  - This allows strict-serializable semantics for reads and writes
- READ_YOUR_WRITES (Kudu 1.7 and up)
  - Ensures all previously read and written values are read
  - Not repeatable


Meta Cache
T1: …
T2: …
T3: …
Client
Hey Master! Where is the row for ‘tlipcon’ 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, …
UPDATE tlipcon
SET col=foo
Master tablet
Master Server A
Master tablet LEADER
Master Server B
Master tablet FOLLOWER
Master Server C
Master tablet FOLLOWER
Tablet 1
Tablet 2
...
Tablet n
Tablet 1 LEADER
Tablet 3 FOLLOWER
Tablet Server W
Tablet 1 FOLLOWER
Tablet 2 FOLLOWER
Tablet Server X
Tablet 1 FOLLOWER
Tablet 2 FOLLOWER
Tablet 3 LEADER
Tablet Server Y
Tablet 2 LEADER
Tablet 3 FOLLOWER
Tablet Server Z

Continuing the walkthrough!

https://bit.ly/2ZsR6m6

- Explore a multinode deployment

Time Series

To demonstrate advanced table partitioning techniques, we are going to think through a table for time series storage of machine metrics.

Series → Time → Value

Time Series

Series	Time	Value
us-east.appserver01.loadavg.1min	2016-05-09T15:14:30Z	0.44
us-east.appserver01.loadavg.1min	2016-05-09T15:14:40Z	0.53
us-west.dbserver03.rss	2016-05-09T15:14:30Z	1572864
us-west.dbserver03.rss	2016-05-09T15:15:00Z	2097152

Time Series — Design Criteria

- Insert Performance (throughput & latency)
- Read Performance (throughput & latency)
  - What kind of queries are you doing?
    - Look at all metrics at a specific time
    - Look at one metric across a long span of time

Time Series — Common Patterns

- Datapoints are inserted **in time order** across all series
- Reads specify a **series** and a **time range**, containing hundreds to many thousands of datapoints

```
SELECT time, value FROM timeseries
WHERE series = "us-west.dbserver03.rss"
  AND time >= 2016-05-08T00:00:00;
```

CLOUDERA

Reminder: Partitioning vs Indexing

- **Partitioning:** how datapoints are distributed among partitions
    - Kudu: tablet
    - HBase: region
    - Cassandra: VNode
- **Indexing:** how data within a single partition is sorted

Reminder: Partitioning vs Indexing

- **Partitioning:** how datapoints are distributed among partitions
  - Kudu: tablet
  - HBase: region
  - Cassandra: VNode
- **Indexing:** how data within a single partition is sorted

(us-east.appserver01.loadavg,	2016-05-09T15:14:00Z)
(us-east.appserver01.loadavg,	2016-05-09T15:15:00Z)
(us-west.dbserver03.rss,	2016-05-09T15:14:30Z)
(us-west.dbserver03.rss,	2016-05-09T15:14:30Z)

(series, time)

```
(2016-05-09T15:14:00Z, us-east.appserver01.loadavg)
(2016-05-09T15:14:30Z, us-west.dbserver03.rss)
(2016-05-09T15:15:00Z, us-east.appserver01.loadavg)
(2016-05-09T15:14:30Z, us-west.dbserver03.rss)
```

(time, series)

Reminder: Partitioning vs Indexing

- **Partitioning:** how datapoints are distributed among partitions
  - Kudu: tablet
  - HBase: region
  - Cassandra: VNode
- **Indexing:** how data within a single partition is sorted

(us-east.appserver01.loadavg, 2016-05-09T15:14:00Z)
(us-east.appserver01.loadavg, 2016-05-09T15:15:00Z)
(us-west.dbserver03.rss, 2016-05-09T15:14:30Z)
(us-west.dbserver03.rss, 2016-05-09T15:14:30Z)

(series, time)

(2016-05-09T15:14:00Z, us-east.appserver01.loadavg)
(2016-05-09T15:14:30Z, us-west.dbserver03.rss)
(2016-05-09T15:15:00Z, us-east.appserver01.loadavg)
(2016-05-09T15:14:30Z, us-west.dbserver03.rss)

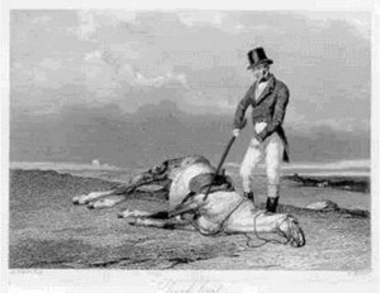
(time, series)

SELECT * WHERE series = ‘us-east.appserver01.loadavg’;

Partitioning

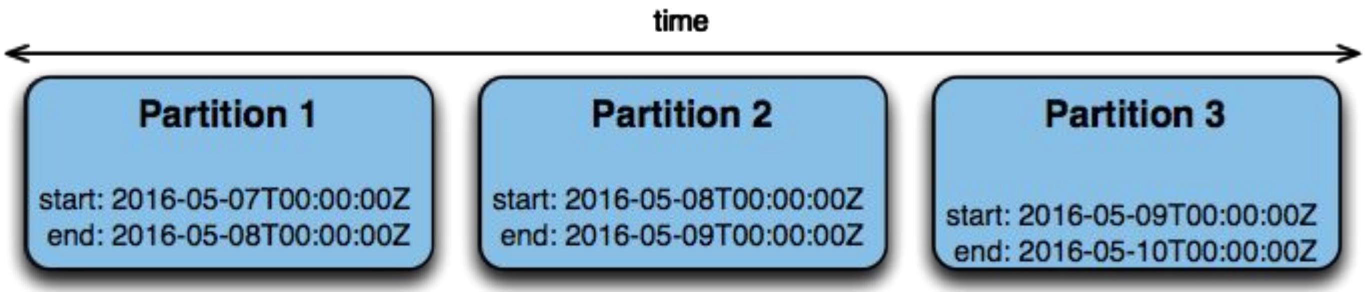
- Kudu has flexible policies for distributing data among partitions
  - Hash partitioning is built in, and can be combined with range partitioning
- **Goal:** acceptable distribution of data across tablets at any given time
- **Goal:** allow expected scans to prune tablets

- Indexing is independent of partitioning!!!

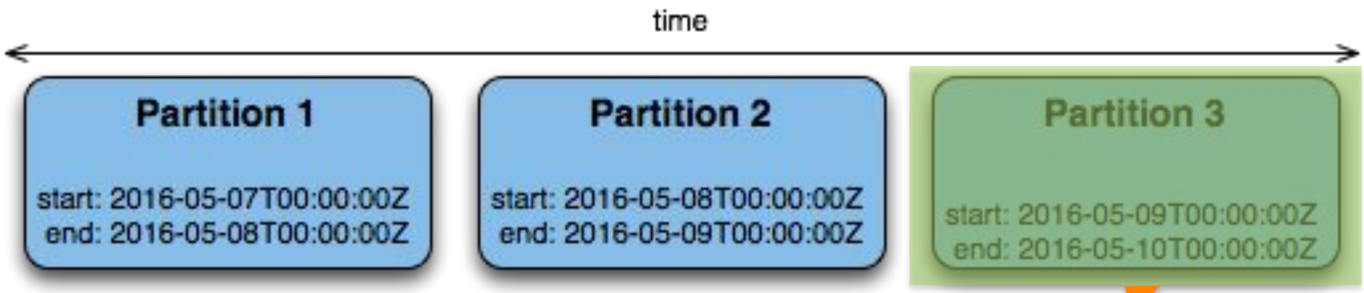


CLOUDERA

Partitioning – By Time Range

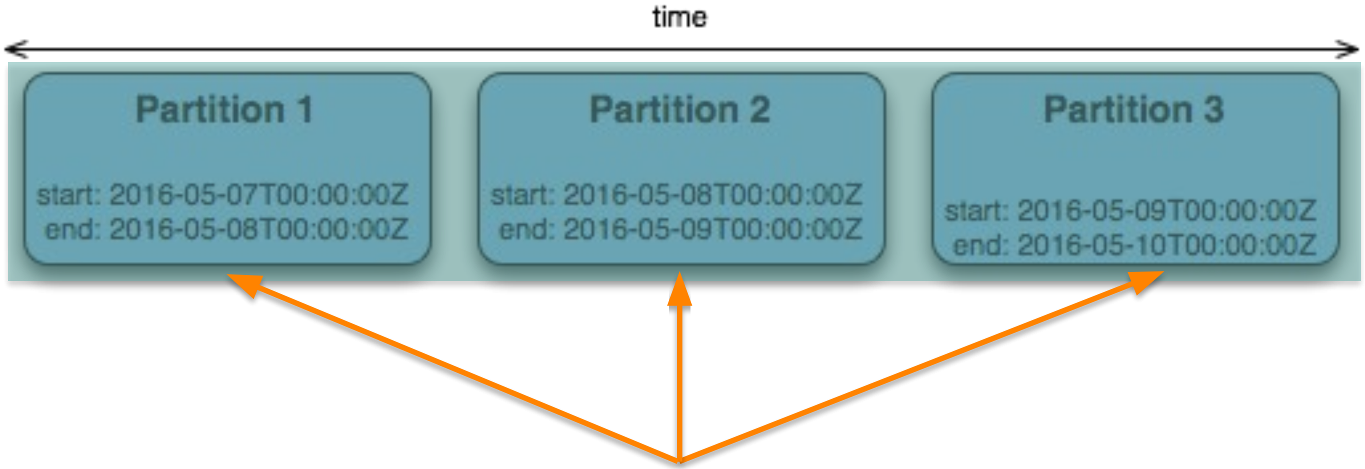


Partitioning – By Time Range (inserts)



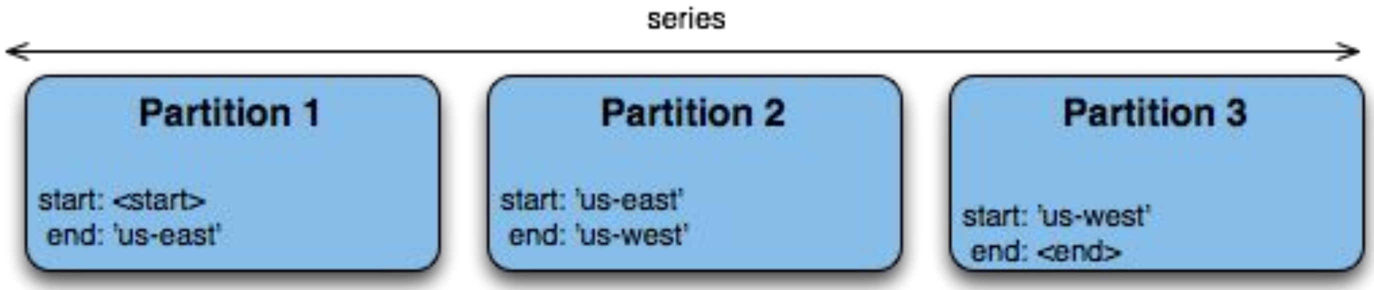
All Inserts go to Latest Partition

Partitioning – By Time Range (scans)

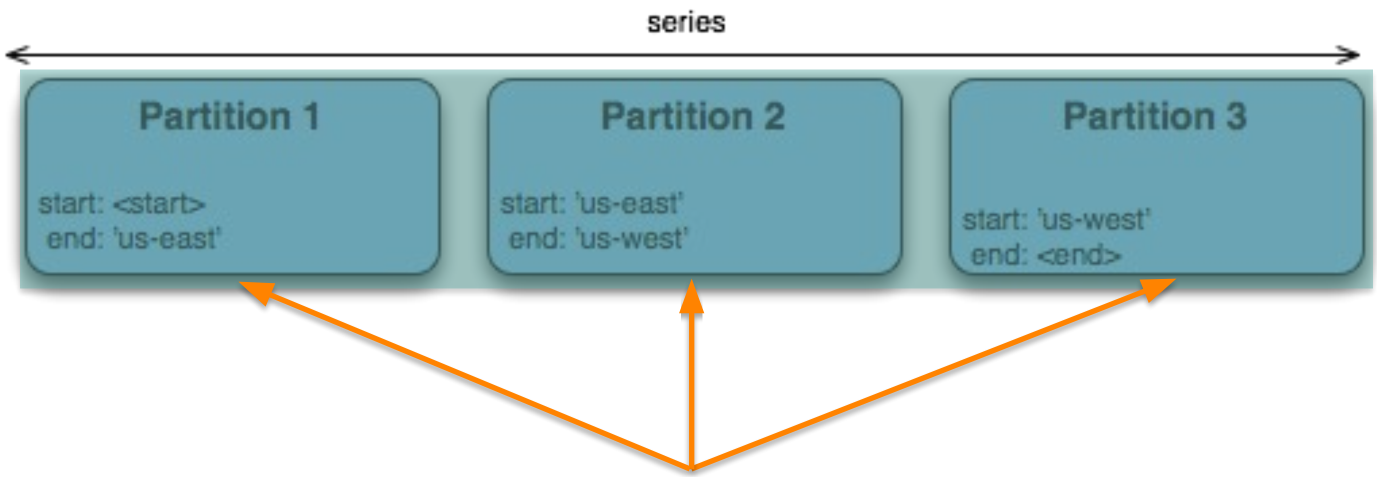


Big scans (across large time intervals)
can be parallelized across many partitions

Partitioning — By Series Range

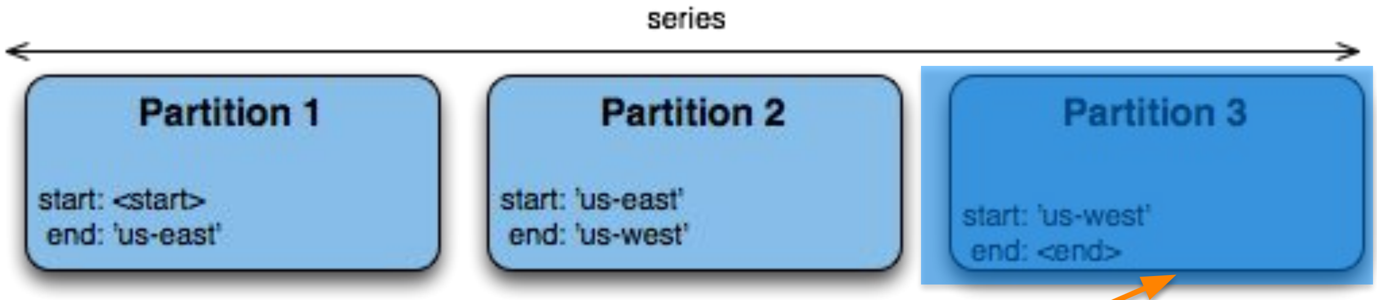


Partitioning — By Series Range (inserts)



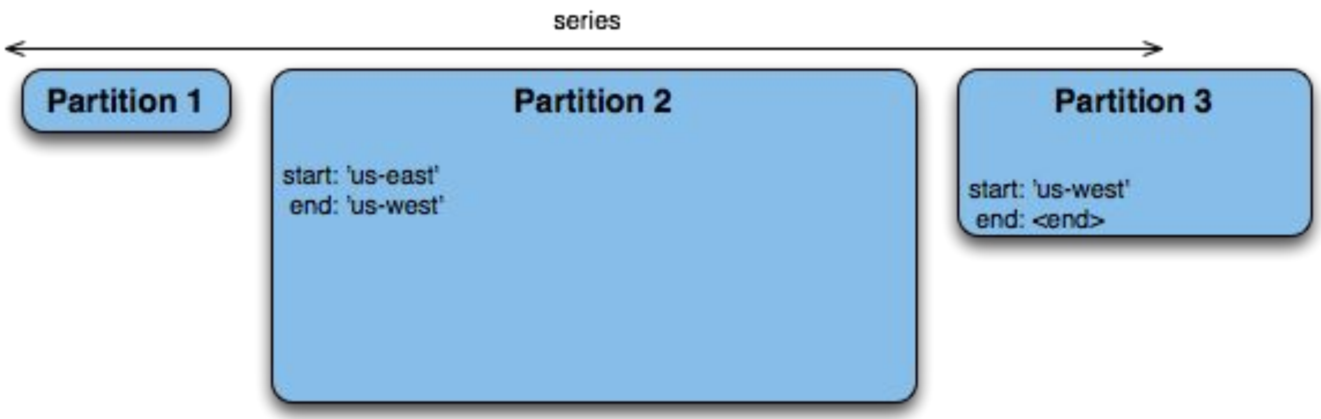
Inserts are spread among all partitions

Partitioning — By Series Range (scans)



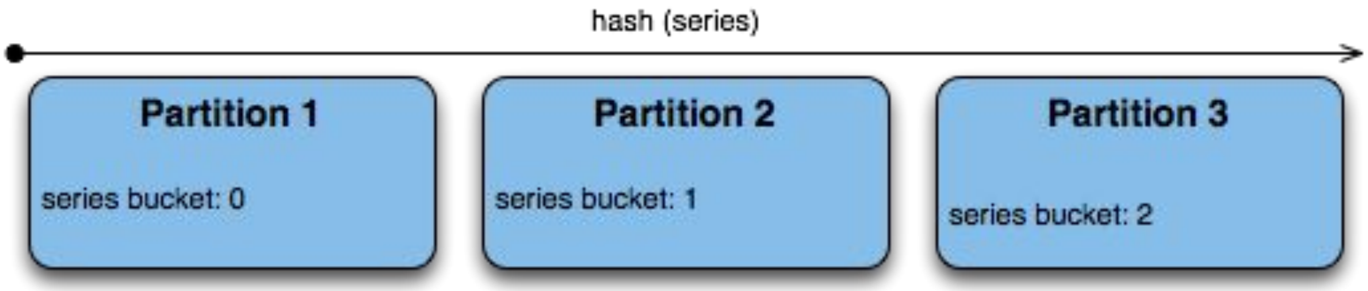
Scans are over a single partition

Partitioning – By Series Range

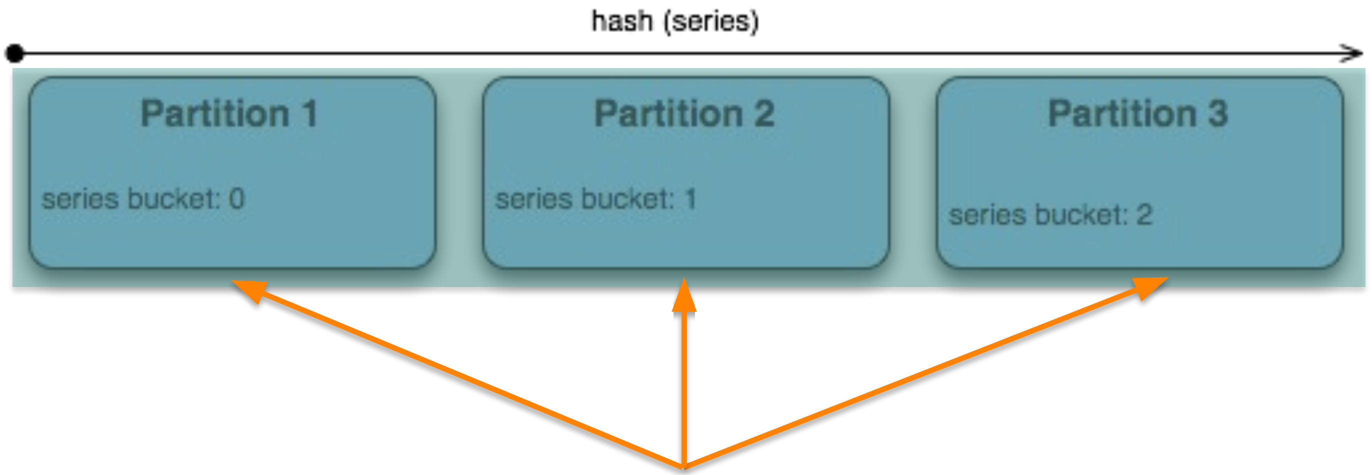


Partitions can become unbalanced,
resulting in hot spotting

Partitioning – By Series Hash

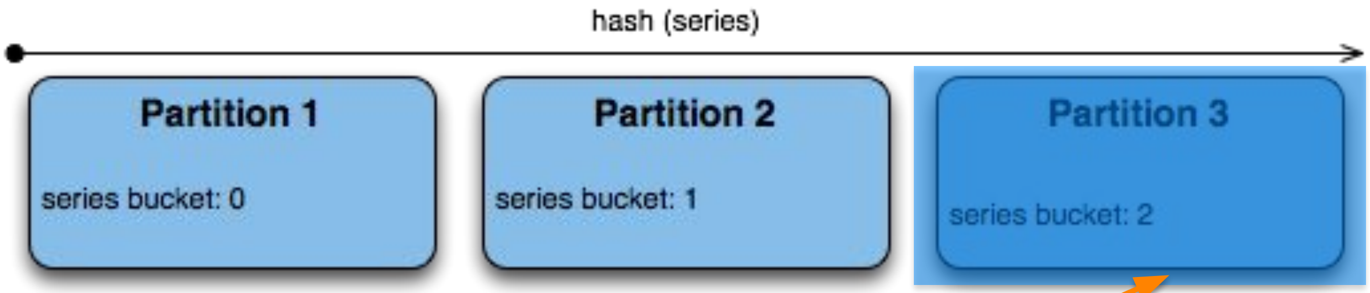


Partitioning – By Series Hash (inserts)



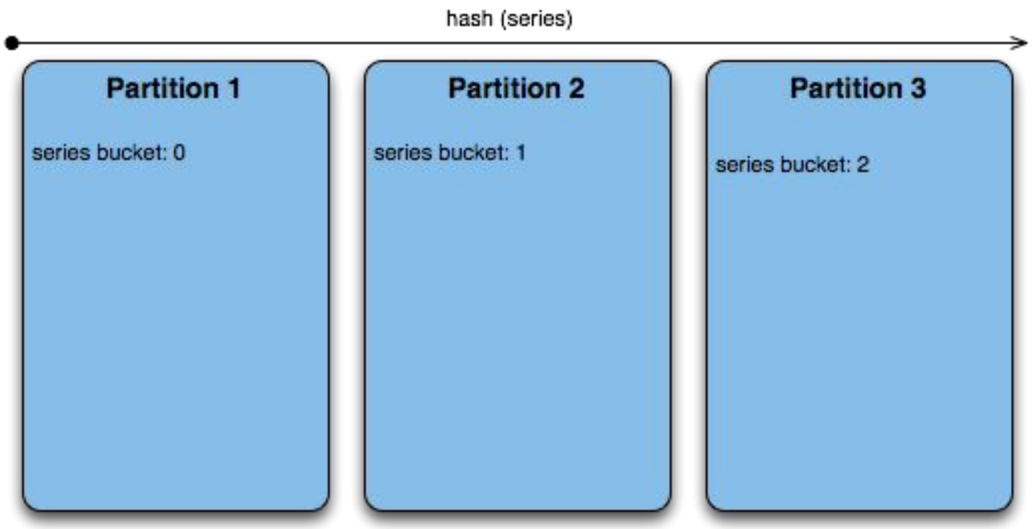
Inserts are spread among all partitions

Partitioning – By Series Hash (scans)



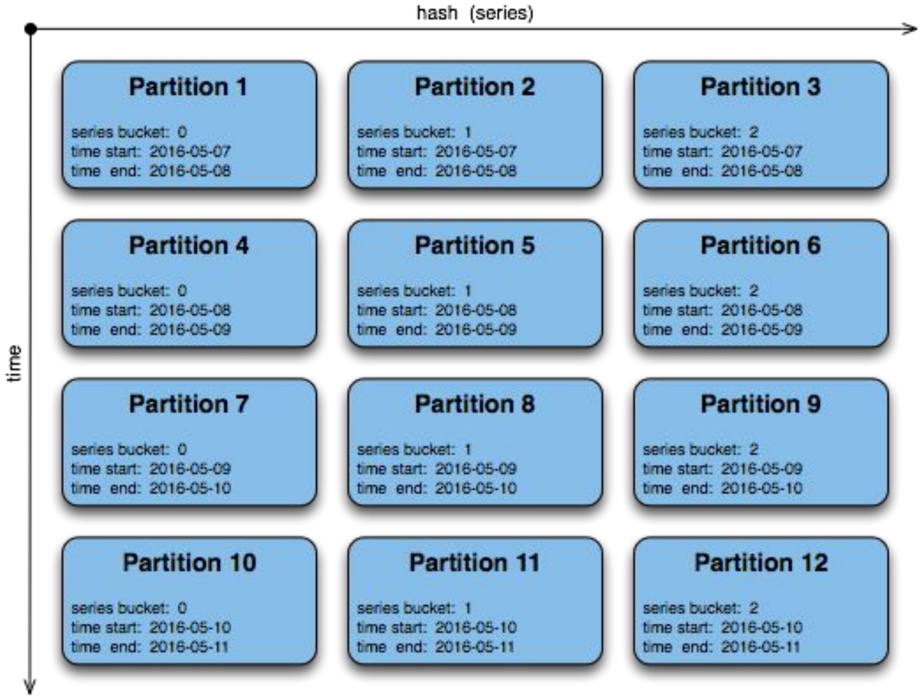
Scans are over a single partition

Partitioning – By Series Hash



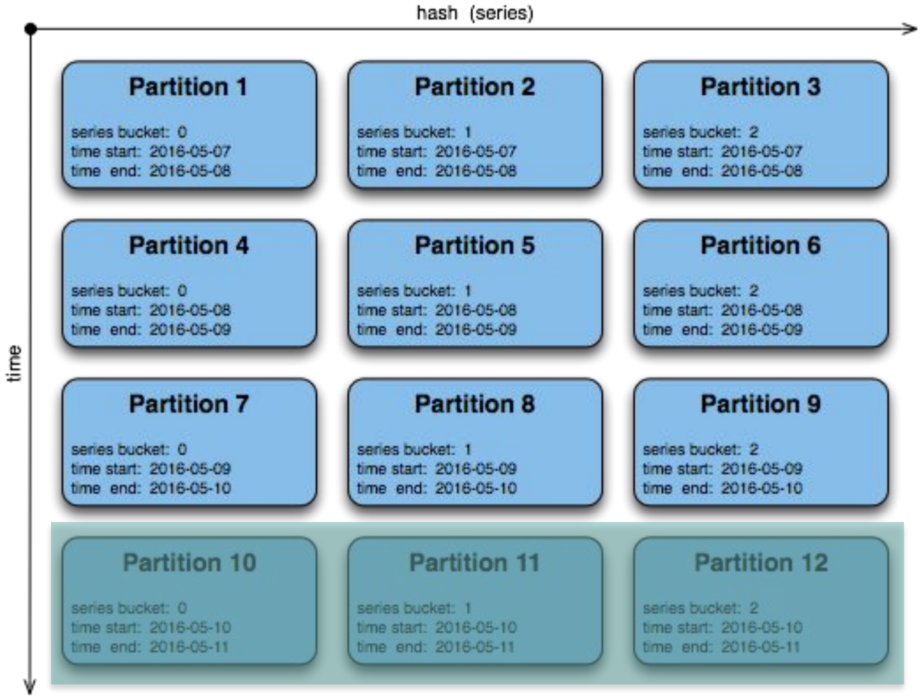
Partitions grow overtime, eventually becoming too big for a single server

Partitioning — By Series Hash + Time Range



CLOUDERA

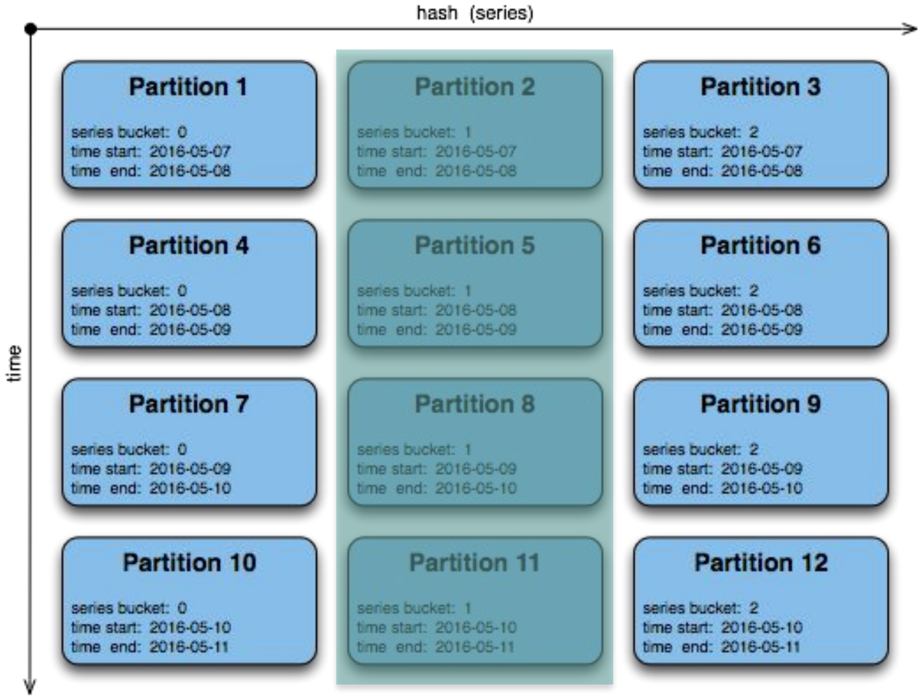
Partitioning – By Series Hash + Time Range (inserts)



Inserts are spread among all partitions
in the latest time range

CLOUDERA

Partitioning – By Series Hash + Time Range (scans)



Big scans (across large time intervals)
can be parallelized across partitions

Spark SQL and Nifi walkthrough

- Deploy some Spark
- Deploy some Nifi
- ???
- Profit

Spark DataSource optimizations

- Column projection and predicate pushdown
  - Only read the referenced columns
  - Convert ‘WHERE’ clauses into Kudu predicates
  - Kudu predicates automatically convert to primary key scans, etc

Spark DataSource optimizations

Predicate pushdown

```
scala> sqlContext.sql("select avg(value) from metrics where host = 'e1103.halxg.cloudera.com'").explain
== Physical Plan ==
TungstenAggregate(key=[], functions=[(avg(value#3),mode=Final,isDistinct=false)], output=[_c0#94])
+- TungstenExchange SinglePartition, None
   +- TungstenAggregate(key=[], functions=[(avg(value#3),mode=Partial,isDistinct=false)],
                        output=[sum#98,count#99L])
      +- Project [value#3]
         +- Scan org.apache.kudu.spark.kudu.KuduRelation@e13cc49[value#3]
                PushedFilters: [EqualTo(host,e1103.halxg.cloudera.com)]
```

CLOUDERA

Spark DataSource optimizations

Partition pruning

```
scala> df.where("host like 'foo%'").rdd.partitions.length
res1: Int = 20
scala> df.where("host = 'foo'").rdd.partitions.length
res2: Int = 1
```

Writing via Spark

```
// Use KuduContext to create, delete, or write to Kudu tables
val kuduContext = new KuduContext("kudu-master:7051,kudu-master:7151,kudu-master:7251")

// Create a new Kudu table from a dataframe schema
// NB: No rows from the dataframe are inserted into the table
kuduContext.createTable("test_table", df.schema, Seq("key"), new CreateTableOptions().setNumReplicas(1))

// Insert, delete, upsert, or update data
kuduContext.insertRows(df, "test_table")
kuduContext.deleteRows(sqlContext.sql("select id from kudu_table where id >= 5"), "kudu_table")
kuduContext.upsertRows(df, "test_table")
kuduContext.updateRows(df.select(“id”, $”count” + 1, "test_table")
```

Spark SQL and Nifi walkthrough

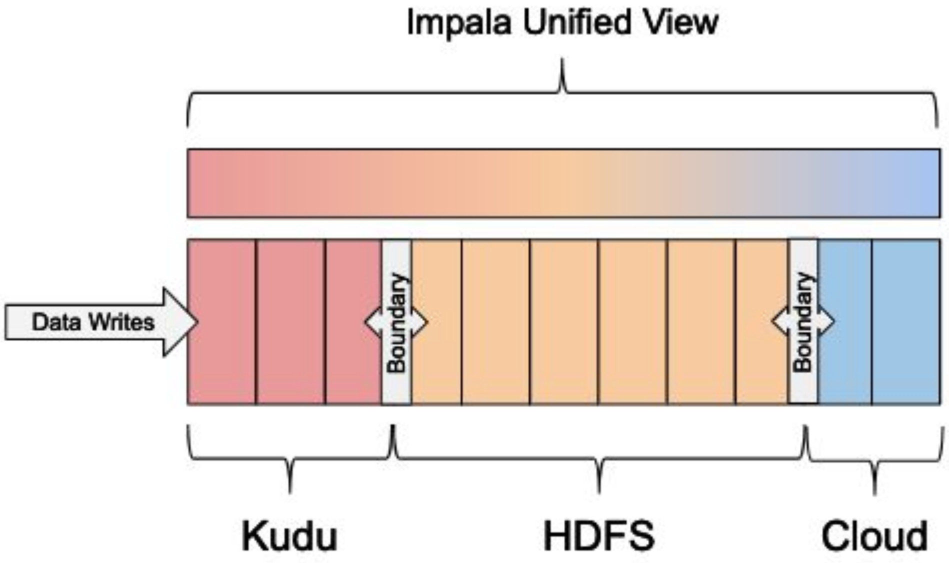
https://bit.ly/2ZsR6m6

- Deploy some Spark
- Deploy some Nifi
- ???
- Profit

Interesting pattern: Hierarchical storage

Return of the Lambda? (but longer time scale)

- Use Impala to periodically (e.g. every month) move data from Kudu into cold storage
- Query both hot Kudu data and colder HDFS and Cloud data with a view
- Simpler primitives than before



Excellent blog post:

https://kudu.apache.org/2019/03/05/transparent-hierarchical-storage-management-with-apache-kudu-and-impala.html

CLOUDERA

Thank you!

Twitter: @ApacheKudu

Slack: https://getkudu-slack.herokuapp.com/

Website: kudu.apache.org

Questions?

awong@cloudera.com