Creating real-time data applications with Impala and Kudu

Marcel Kornacker (Impala founder)
Todd Lipcon (Kudu founder)
Software Engineers at Cloudera
Hadoop Architecture Overview
One Platform, Multiple Components
This talk: relational data management in Hadoop
Choosing the right storage for relational data
Many choices to mix and match

- Apache HDFS
- Apache HBase
- Apache Kudu
“Traditional” Analytics in Hadoop
Fastest analytics on archived data with HDFS + Impala

**Flexibility** to store any type of data in any format

**Infinite scalability** for cost-effective active archival

**Highest throughput and storage density** for analytics on *static* data sets

**Limited/no ability for updates, deletes, or streaming inserts**
"Traditional" On-Line Data Management in Hadoop
Storage for random read/write access, large scale serving

**Flexibility** to store any type of data with semi-structured schema (but difficult to query with SQL)

**Real-Time Data Ingest and Serving**
- Built to handle fast changing data
- Serve data at scale

**Poor performance for analytic queries**
Traditional Hadoop Storage Leaves a Gap
Use cases that fall between HDFS and HBase were difficult to manage

Real-Time
Fast Changing
Frequent Updates
Append-Only
Unchanging

Pace of Data

Arbitrary Storage
(Append-Only)

HDFS

Pace of Analysis

Fast Scans, Analytics
and Processing of
Static Data

Fast Analytics
(on fast-changing or
frequently-updated data)

Fast On-Line
Updates &
Data Serving

Analytic
Gap

Complex Hybrid
Architectures

HBase
Kudu: Fast Analytics on Fast-Changing Data
New storage engine enables new Hadoop use cases

HDFS
Fast Scans, Analytics and Processing of Static Data

Kudu
Fast Analytics (on fast-changing or frequently-updated data)

HBase
Fast On-Line Updates & Data Serving

Kudu fills the Gap
Modern analytic applications often require complex data flow & difficult integration work to move data between HBase & HDFS

Arbitrary Storage (Active Archive)

Real-Time
Fast Changing Frequent Updates
Append-Only
Unchanging

Pace of Data
Pace of Analysis

© Cloudera, Inc. All rights reserved.
Online Analytics in Hadoop
Simple real-time analytics and updates with Apache Kudu

**Simplified architecture** for building real-time analytic applications

**Fast Analytics on Fast Data**
- Reporting with update support through Kudu and Impala
- Real-time and streaming applications with Kudu + Spark
Apache Impala (incubating)
High performance SQL on Hadoop
Impala: A Modern, Open-Source SQL Engine

• Implementation of an MPP SQL engine for the Hadoop environment
• Designed for performance: brand-new engine written in C++
• Maintains Hadoop flexibility by utilizing standard Hadoop components:
  • storage managers: Kudu, HDFS, HBase
  • metadata: MetaStore
  • reads widely used file formats: Parquet, text, Avro, …
  • runs on same nodes that run Hadoop processes
• Apache Incubating
• Started in 2011, released in beta in 10/2012, 1.0 in 05/2013
Apache Impala (Incubating): Open Source & Open Standard

1. > 1 MM downloads since GA

2. Majority adoption across Cloudera customers

3. Certification across key application partners:
   - MicroStrategy
   - Qlik
   - SAS
   - Tableau
   - SAP
   - IBM Cognos
   - Microsoft
   - Oracle
   - and others

4. De facto standard with multi-vendor support:
   - Cloudera
   - MapR
   - Oracle
Impala from the User’s Perspective

• Designed to play well with BI tools
• Standard ANSI SQL (92, with 2003 analytic extensions), UDFs/UDAs, correlated subqueries, nested types, …
• Data types:
  • Integer and floating point types, STRING, CHAR, VARCHAR, TIMESTAMP
  • DECIMAL(<precision>, <scale>) with up to 38 digits of precision
• Connect via odbc/jdbc
• Authenticate via Kerberos/LDAP
• Authorization with GRANT/REVOKE
Impala: highest performance SQL on Hadoop

- Multi-user workload
- 66 unmodified TPC-DS queries
- 8 independent query streams
- Impala outperforms Spark-SQL 2.0 by 9x & Presto by 23x
- 45% of the queries succeed with Spark-SQL, the remaining queries error out
- Only 66% of the queries succeed with Presto, the remaining queries error out
Apache Kudu usage and design
Apache Kudu: Scalable and fast tabular storage

Scalable
• Tested up to 275 nodes (~3PB cluster)
• Designed to scale to 1000s of nodes and tens of PBs

Fast
• Millions of read/write operations per second across cluster
• Multiple GB/second read throughput per node

Tabular
• Represents data in structured tables like a relational database
• Individual record-level access to 100+ billion row tables
Storing records in Kudu tables

• A Kudu table has a **SQL-like schema**
  • And a **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, Python, and C++ **NoSQL-style APIs**
  • Insert(), Update(), Delete(), Scan(), low-milliseconds latencies
• **SQL** via integrations with Impala and Spark
Physical schema design in Kudu

Example: Time Series Data
What is time series?

Data that can be usefully partitioned and queried based on time

Examples:

- Web user activity data (view and click data, tweets, likes)
- Machine metrics (CPU utilization, free memory, requests/sec)
- Patient data (blood pressure readings, weight changes over time)
- Financial data (stock transactions, price fluctuations)
Kudu and time series data

Real time data ingestion + fast scans = Ideal platform for storing and querying time series data

• Support for many column encodings and compression schemes
  • Encodings: Delta, dictionary, bitshuffle
  • Compression: LZ4, gzip, bzip2
• Kudu supports a flexible range of partitioning schemes
  • Partition by time range, hash, or both
• Parallelizable scans
• Scale-out storage system
Partitioning by time range + series hash
Partitioning by time range + series hash (inserts)

Inserts are spread among all partitions of the time range
Partitioning by time range + series hash (scans)

Big scans (across time intervals) can be parallelized across partitions
Kudu and Impala together
Anatomy of a Data Management System

Monolithic RDBMS

- Query Execution
- Record Layer
- Storage

RDBMS functionality in Hadoop

- Impala
- Parquet
- Kudu
- HDFS

© Cloudera, Inc. All rights reserved.
Advantages of a decoupled architecture

• **Mix and match access mechanisms** against single storage manager
  • Impala for SQL
  • Spark for machine learning
  • “NoSQL” APIs for low-latency random access

• **Mix and match storage managers** within a single application (or query)
  • `SELECT COUNT(*) FROM my_fact_table_on_hdfs JOIN my_dim_table_in_kudu ON ...`

• **Preserve same SQL support** with new storage managers
  • BI tools (Tableau, ZoomData, etc) work automatically with Kudu tables
Impala makes full administrative functionality available via SQL

- simple statements to create, drop, and alter tables
- works for tables created with Impala and tables created initially via API

```
CREATE TABLE metrics (  
    host STRING,  
    metric STRING,  
    ts INT64,  
    value DOUBLE,  
    PRIMARY KEY(host, metric, ts)  
) DISTRIBUTED BY HASH(metric) INTO 3 BUCKETS,  
RANGE(ts) SPLIT ROWS ((...), (...))  
STORED AS KUDU;
```
SQL Syntax and Examples: SELECT

Impala takes advantage of the features of the scan API:

- tablets are selected based on supplied key predicates
- Kudu-only predicates are pushed down
- remaining predicates are evaluated in Impala

Impala and Kudu share a common in-memory data representation:
- avoids conversion overhead

```
select *
from metrics
where ts > 1475078450
and hostname like '%foo.com'
and metric = 'cpu.loadavg'

---- PLAN
00:SCAN KUDU [example.metrics]
  predicates: hostname like '%foo.com'
  kudu predicates: timestamp > 1475078450, metric = 'cpu.loadavg'
---- SCANRANGELOCATIONS
NODE 0:
  ScanToken{table=metrics,
              range-partition: [(int64 ts=1475000000), (int64 ts=1480000000)],
              hash bucket: 3}
```
SQL Syntax and Examples: INSERT

A single-row insert is translated into a single API call.

```
insert into metrics(host, metric, ts, value) values
("foo.example.com", "cpu.loadavg", 1475078450, 1.283);
```

```
INSERT INTO KUDU [example.metrics]
| 00:UNION
constant-operands=1
```
SQL Syntax and Examples: UPDATE

- UPDATE statements can be arbitrarily complex
- .. and can reference non-Kudu tables to generate the key set and updated values

```
update a
set a.value = ...
from example.metrics a
join example.hostinfo b
on a.hostname = b.hostname
where b.datacenter = 'eastcoast';
```

--- PLAN
UPDATE KUDU [example.metrics]
|
02:HASH JOIN [INNER JOIN]
|  hash predicates: a.hostname = b.hostname
|
|--01:SCAN HDFS [example.hostinfo b]
|    partitions=1/1 files=0 size=0B
|    predicates: b.datacenter = 'eastcoast'
|
00:SCAN KUDU [example.metrics a]
SQL Syntax and Examples: DELETE

- DELETE statements can also be arbitrarily complex
- the key set is determined by running a query
- takes full advantage of existing optimizations, such as runtime filters

```sql
delete a
from example.metrics a
join example.hostinfo b
on a.hostname = b.hostname
where b.datacenter = 'eastcoast';
```

---- PLAN
DELETE FROM KUDU [example.metrics]
| 02:HASH JOIN [INNER JOIN]
|   hash predicates: a.hostname = b.hostname
|   01:SCAN HDFS [example.hostinfo b]
|   partitions=1/1 files=0 size=0B
|   predicates: b.datacenter = 'eastcoast'
| 00:SCAN KUDU [example.metrics a]
Application architectures
Goals for “real time” applications

- Continuously load data
  - all the time, not once a day/hour/etc

- Visible with minimal delay
  - queries reflect up-to-date data

- Consistent semantics
  - Data once seen doesn’t disappear
  - Data shows up exactly once
“Traditional” real-time analytics in Hadoop

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?

Storage in HDFS

- Historical Data
- Most Recent Partition
- New Partition

Kafka

Have we accumulated enough data?

Avro

Convert Avro file into Parquet

Parquet File

- Wait for running operations to complete
- Define new Impala partition referencing the newly written Parquet file

Reporting Request

Storage in HDFS

- Historical Data
- Most Recent Partition
- New Partition

Kafka

Have we accumulated enough data?

Avro

Convert Avro file into Parquet

Parquet File

- Wait for running operations to complete
- Define new Impala partition referencing the newly written Parquet file

Reporting Request
Real-time analytics in Hadoop with Impala and Kudu

Improvements:
- One system to operate
- No cron jobs or background processes
- Handle late arrivals or data corrections with ease
- New data available immediately for analytics or operations

Incoming streaming data (e.g. Kafka or NoSQL APIs)
Data loading via SQL
Storage in Kudu
Historical and Real-time Data
SQL Reporting
Spark ML modeling
Kudu+Impala vs MPP DWH

Commonalities
✓ Fast analytic queries via SQL, including most commonly used modern features
✓ Ability to insert, update, and delete data

Differences
✓ Faster streaming inserts
✓ Improved Hadoop integration
  • JOIN between HDFS + Kudu tables, run on same cluster
  • Spark, Flume, other integrations
✗ Slower batch inserts
✗ No transactional data loading, multi-row transactions, or indexing
Summary
Summary

- Kudu and Impala together provide
  - **RDBMS-like capabilities** for big data: INSERT, UPDATE, and DELETE
  - **Simplified architecture** that combines analytics and streaming ingest
- **Decoupled architecture** maintains flexibility of the Hadoop stack
  - Access the same data store from multiple engines (Impala, Spark, etc)
  - Single SQL view spans multiple storage engines (Kudu, Parquet, etc)
- **High performance architecture** combines MPP-like features with Hadoop’s scalability
  - MPP query execution
  - Flexible physical schema design for maximum performance
  - Scalability to hundreds of nodes and high concurrency
Backup slides
Kudu Increases the Value of Time Series Data

Time series data is most valuable if you can analyze it to change outcomes in real time.

Kudu simultaneously enables:
• Time series data inserted/updated as it arrives
• Analytic scans to find trends on fresh time series data
• Lookups to quickly visit the point in time where an event occurred for further investigation

Examples
Stream market data, fraud detection & prevention, risk monitoring

Workload
Inserts, updates, scans, lookups
Kudu can help spot problems before they happen. Real-time data inserts with the ability to analyze trends identifies potential problems.

Kudu identifies trouble through:
- Extreme scale, allowing better historic trend analysis
- Fast inserts to enable an up-to-date view of your business
- Fast scans identify/flag undesired states for remedy

Examples
Network threat detection, IoT, predictive maintenance and failure detection
Workload
Inserts, scans, lookups
More Versatility in Online Reporting

Online reporting has traditionally been limited by data volume and analytic capability, keeping only recent data designed for granular queries.

Kudu adds online reporting versatility through:
- Fast inserts and updates to keep data fresh
- Fast lookups and analytic scans in one data store

Examples
“Active” Reporting

Workload
Inserts, updates, scans, lookups
How it works
Client metadata lookup
Kudu Master: Keeper of metadata

Replicated master

• Acts as a tablet directory
• Acts as a catalog (which tables exist, etc)
• Acts as a load balancer (tracks TS liveness, re-replicates under-replicated tablets)

Not a bottleneck

• Super fast in-memory lookups
Hey Master! Where is the row for ‘mpercy’ 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 mpercy
SET col=foo
How it works
Write and read paths
Kudu storage – Tablet internals

• 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
Performance
TPC-H (analytics benchmark)

• 75 server cluster
  • 12 (spinning) disks each, enough RAM to fit dataset
  • TPC-H Scale Factor 100 (100GB)
• Example SQL query (via Impala):
  • 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;
TPC-H results: Kudu vs Parquet

- Kudu outperforms Parquet by 31% (geometric mean) for RAM-resident data
TPC-H results: Kudu vs other NoSQL storage

Apache Phoenix: OLTP SQL engine built on HBase

- 10 node cluster (9 workers, 1 master)
- TPC-H LINEITEM table only (6B rows)
What about NoSQL-style random access? (YCSB)

- YCSB 0.5.0-snapshot
- 10 node cluster (9 workers, 1 master)
- 100M row data set
- 10M operations each workload
Xiaomi benchmarks
Write and read paths
Xiaomi benchmark

- 6 real queries from application trace analysis application
  - Q1: SELECT COUNT(*)
  - Q2: SELECT hour, COUNT(*) WHERE module = ‘foo’ GROUP BY HOUR
  - Q3: SELECT hour, COUNT(DISTINCT uid) WHERE module = ‘foo’ AND app=‘bar’ GROUP BY HOUR
  - Q4: analytics on RPC success rate over all data for one app
  - Q5: same as Q4, but filter by time range
  - Q6: SELECT * WHERE app = … AND uid = … ORDER BY ts LIMIT 30 OFFSET 30
Xiaomi benchmark results

Query latency (seconds):

- HDFS parquet file replication = 3
- Kudu table replication = 3
- Each query run 5 times then averaged
Contact Cloudera- Local Points of Contact

Denver Area:

Lisa McGuire- 720.256.4780
Cole Waldron- 415.815.8212
Chris Cornacchia- 303.579.6312
Next generation hardware

Solid-state Storage
Cheaper and faster every year.
Persistent memory (3D XPoint™)

Kudu can take advantage of SSD and NVM using Intel’s NVM Library.

Cheaper, Bigger Memory
RAM is cheaper and bigger every day.

Kudu runs smoothly with huge RAM. Written in C++ to avoid GC issues.

Efficiency on Modern CPUs
Modern CPUs are adding cores and SIMD width, not GHz.

Kudu takes advantage of SIMD instructions and concurrent data structures.
## Columnar storage

### Twitter Firehose Table

<table>
<thead>
<tr>
<th>tweet_id</th>
<th>user_name</th>
<th>created_at</th>
<th>text</th>
</tr>
</thead>
<tbody>
<tr>
<td>22309487</td>
<td>RidelImpala</td>
<td>1442828307</td>
<td>Introducing the Ibis project: for the Python experience at Hadoop Scale</td>
</tr>
<tr>
<td>23059861</td>
<td>fastly</td>
<td>1442865156</td>
<td>Missed July's SF @papers_we_love? You can now watch @el_bhs talk about @google's globally-distributed database: <a href="http://fastly.us/1eVz8MM">http://fastly.us/1eVz8MM</a></td>
</tr>
<tr>
<td>23010982</td>
<td>llvmorg</td>
<td>1442865155</td>
<td>LLVM 3.7 is out! Get it while it's HOT! <a href="http://llvm.org/releases/download.html#3.7.0">http://llvm.org/releases/download.html#3.7.0</a></td>
</tr>
</tbody>
</table>

### Diagram

- **Tweet_id**
  - `{25059873, 22309487, 23059861, 23010982}`

- **User_name**
  - `{newsycbot, RidelImpala, fastly, llvmorg}`

- **Created_at**
  - `{1442865158, 1442828307, 1442865156, 1442865155}`

- **text**
  - `{Visual exp..., Introducing .., Missing July..., LLVM 3.7....}`
SELECT COUNT(*) FROM tweets WHERE user_name = 'newsycbot';
**Columnar compression**

- Many columns can compress to a few bits per row!
- Especially:
  - Timestamps
  - Time series values
  - Low-cardinality strings
- Massive space savings and throughput increase!

<table>
<thead>
<tr>
<th>Created_at</th>
<th>Diff(created_at)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1442825158</td>
<td>n/a</td>
</tr>
<tr>
<td>1442826100</td>
<td>942</td>
</tr>
<tr>
<td>1442827994</td>
<td>1894</td>
</tr>
<tr>
<td>1442828527</td>
<td>533</td>
</tr>
<tr>
<td>64 bits each</td>
<td>11 bits each</td>
</tr>
</tbody>
</table>
**Why Kudu?**
A simultaneous combination of sequential and random reads and writes

---

**Time Series Data**
Can you insert time series data in real time? How long does it take to prepare it for analysis? Can you get results and act fast enough to change outcomes?

---

**Machine Data Analytics**
Can you handle large volumes of machine-generated data? Do you have the tools to identify problems or threats? Can your system do machine learning?

---

**Online Reporting**
How fast can you add data to your data store? Are you trading off the ability to do broad analytics for the ability to make updates? Are you retaining only part of your data?
Getting started with Kudu
Project status

• 1.0 release last week!

• Usable for many applications (several companies are running in production)
  • Have not experienced unrecoverable data loss.
  • Users deploying up to 200 nodes so far
  • Some important features still in progress (security)

• Kudu is a top-level project (TLP) at the Apache Software Foundation
  • Community-driven open source process
Getting started as a user

• On the web: kudu.apache.org
• User mailing list: user@kudu.apache.org
• Slack chat channel (see web site)

• Quickstart VM
  • Easiest way to get started
  • Impala and Kudu in an easy-to-install VM
• CSD and Parcels
  • For installation on a Cloudera Manager-managed cluster
Getting started as a developer

• Source code: github.com/apache/kudu
• Code reviews: gerrit.cloudera.org
• Public JIRA: issues.apache.org/jira/browse/KUDU
• Developer mailing list: dev@kudu.apache.org

• Apache 2.0 license open source and an ASF project
• Contributions welcome and encouraged!
Tables, tablets, and tablet servers

- Each 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
  - Automatic **fault tolerance**
  - MTTR: ~5 seconds
- **Tablet servers** host tablets on local disk drives
- **HA Masters** service metadata operations
  - Create/drop tables and tablets
  - Locate tablets
Kudu-TS library (early stages)

Created by Dan Burkert (Kudu PMC, Cloudera)

- A library and set of Java APIs for storing and querying metrics data in Kudu
- Write time series data points
- Query time series data
  - Optimized for queries that specify a metric and set of tag filters
  - Aggregate, downsample and interpolate results with per-query policies
- Provides a flexible, high-performance data model out of the box
- Code: github.com/danburkert/kudu-ts