Apache Kudu
A Distributed, Columnar Data Store for Fast Analytics

Mike Percy
Software Engineer at Cloudera
Apache Kudu PMC member
Kudu Overview
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

HDFS

Complex Hybrid Architectures

- Fast Analytics (on fast-changing or frequently-updated data)
- Fast On-Line Updates & Data Serving

© Cloudera, Inc. All rights reserved.
Kudu: Fast Analytics on Fast-Changing Data
New storage engine enables new Hadoop use cases

- Real-Time
- Fast Changing, Frequent Updates
- Append-Only
- Unchanging

<table>
<thead>
<tr>
<th>HDFS</th>
<th>Kudu</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arbitrary Storage (Active Archive)</td>
<td>Fast Scans, Analytics and Processing of Static Data</td>
</tr>
</tbody>
</table>

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

**Pace of Analysis**
- Fast On-Line Updates & Data Serving
- Fast Analytics (on fast-changing or frequently-updated data)
Apache Kudu: Scalable and fast tabular storage

Tabular
• Represents data in structured tables like a relational database
  • Strict schema, finite column count, no BLOBs
• Individual record-level access to 100+ billion row tables

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
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()
- SQL via integrations with Spark and Impala
  - Community work in progress / experimental: Drill, Hive
Kudu SQL access

• Kudu itself is just storage and native “NoSQL” APIs

• SQL access via integrations with Spark, Impala, etc.
Kudu “NoSQL” APIs - Writes

```java
KuduTable table = client.openTable(“my_table”);
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();
```
Kudu “NoSQL” APIs - Reads

```java
KuduScanner scanner = client.newScannerBuilder(table)
    .setProjectedColumnNames(List.of("value"))
    .build();
while (scanner.hasMoreRows()) {
    RowResultIterator batch = scanner.nextRows();
    while (batch.hasNext()) {
        RowResult result = batch.next();
        System.out.println(result.getDouble("value");
    }
}
```
Kudu “NoSQL” APIs - Predicates

```java
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.
Tables and tablets

• Each table is horizontally partitioned into tablets
  • Range or hash partitioning
    • PRIMARY KEY (host, metric, timestamp) DISTRIBUTE BY HASH(timestamp) INTO 100 BUCKETS
    • Translation: bucketNumber = hashCode(row['timestamp']) % 100
• Each tablet has N replicas (3 or 5), kept consistent with Raft consensus
• Tablet servers host tablets on local disk drives
Fault tolerance

• Operations replicated using **Raft consensus**
  • Strict quorum algorithm. See Raft paper for details

• Transient failures:
  • Follower failure: Leader can still achieve majority
  • Leader failure: **automatic leader election** (~5 seconds)
  • Restart dead TS within 5 min and it will rejoin transparently

• Permanent failures
  • After 5 minutes, automatically creates a new follower replica and copies data

• **N replicas** can tolerate up to \((N-1)/2\) failures
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)
• Caches all *metadata* in RAM for high performance
• Client configured with master addresses
  • Asks master for tablet locations as needed and caches them
Integrations

Kudu is designed for integrating with higher-level compute frameworks

Integrations exist for:
• Spark
• Impala
• MapReduce
• Flume
• Drill
Apache Kudu
Storage for fast (low latency) analytics on fast (high throughput) data

- Simplifies the architecture for building analytic applications on changing data
- Optimized for fast analytic performance
- Natively integrated with the Hadoop ecosystem of components
Use cases
Kudu use cases

Kudu is best for use cases requiring:
- Simultaneous combination of sequential and random reads and writes
- Minimal to zero data latencies

Time series
- Examples: Streaming market data; fraud detection & prevention; network monitoring
- Workload: Inserts, updates, scans, lookups

Online reporting / data warehousing
- Example: Operational Data Store (ODS)
- Workload: Inserts, updates, scans, lookups
“Traditional” real-time analytics in Hadoop
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?

Kafka

Have we accumulated enough data?

HBase

Reorganize HBase file into Parquet

Parquet File

Storage in HDFS

Historical Data

Most Recent Partition

New Partition

Reporting Request

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

Have we accumulated enough data?

Reorganize HBase file into Parquet

Parquet File

Storage in HDFS

Historical Data

Most Recent Partition

New Partition

Reporting Request

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

Kafka

Have we accumulated enough data?

HBase

Reorganize HBase file into Parquet

Parquet File

Storage in HDFS

Historical Data

Most Recent Partition

New Partition

Reporting Request

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

Kafka

Have we accumulated enough data?

HBase

Reorganize HBase file into Parquet

Parquet File

Storage in HDFS

Historical Data

Most Recent Partition

New Partition

Reporting Request

• Wait for running operations to complete
• Define new Impala partition referencing the newly written Parquet file
Xiaomi use case

- World’s 4th largest smart-phone maker (most popular in China)
- Gather important RPC tracing events from mobile app and backend service.
- Service monitoring & troubleshooting tool.

High write throughput
- >20 Billion records/day and growing

Query latest data and quick response
- Identify and resolve issues quickly

Can search for individual records
- Easy for troubleshooting
Xiaomi big data analytics pipeline
Before Kudu

Long pipeline
- High data latency (approx 1 hour – 1 day)
- Data conversion pains

No ordering
- Log arrival (storage) order is not exactly logical order
- Must read 2 – 3 days of data to get all of the data points for a single day
Xiaomi big data analytics pipeline
Simplified with Kafka and Kudu

ETL pipeline
- 0 – 10s data latency
- Apps that need to avoid backpressure or need ETL

Direct pipeline (no latency)
- Apps that don’t require ETL or backpressure handling
Real-time analytics in Hadoop with 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
JD.com use case

• 2\textsuperscript{nd} largest online retailer in China
• Real-time ingestion via Kafka
  • Click logs
  • Application/Browser tracing
• ~70 columns per row
• 6/18 sale day
  • 15B transactions
  • 10M inserts/sec peak
• 200 node cluster
• Query via JDBC -> Impala -> Kudu
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
How it works

Columnar storage
## 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>RideImpala</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>1442865156</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, RideImpala, fastly, llvmorg}`
- **Created_at**: `{1442865158, 1442828307, 1442865156, 1442865155}`
- **text**: `{Visual exp..., Introducing .., Missing July..., LLVM 3.7...}`
Columnar storage

SELECT COUNT(*) FROM tweets WHERE user_name = 'newsycbot';
### Columnar compression

#### Created_at

<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>
</tbody>
</table>

- 64 bits each
- 11 bits each

- Many columns can compress to a few bits per row!
- Especially:
  - Timestamps
  - Time series values
  - Low-cardinality strings
- Massive space savings and throughput increase!
Representing time series in Kudu
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 & 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
  • Dynamic partition management: Add / Drop partitions at any time
• Parallelizable scans
• Scale-out storage system
Partitioning by time range + series hash

Partition 1
- Series bucket: 0
- Time start: 2016-05-07
- Time end: 2016-05-08

Partition 2
- Series bucket: 1
- Time start: 2016-05-07
- Time end: 2016-05-08

Partition 3
- Series bucket: 2
- Time start: 2016-05-07
- Time end: 2016-05-08

Partition 4
- Series bucket: 0
- Time start: 2016-05-08
- Time end: 2016-05-09

Partition 5
- Series bucket: 1
- Time start: 2016-05-08
- Time end: 2016-05-09

Partition 6
- Series bucket: 2
- Time start: 2016-05-08
- Time end: 2016-05-09

Partition 7
- Series bucket: 0
- Time start: 2016-05-09
- Time end: 2016-05-10

Partition 8
- Series bucket: 1
- Time start: 2016-05-09
- Time end: 2016-05-10

Partition 9
- Series bucket: 2
- Time start: 2016-05-09
- Time end: 2016-05-10

Partition 10
- Series bucket: 0
- Time start: 2016-05-10
- Time end: 2016-05-11

Partition 11
- Series bucket: 1
- Time start: 2016-05-10
- Time end: 2016-05-11

Partition 12
- Series bucket: 2
- Time start: 2016-05-10
- Time end: 2016-05-11
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
Integrations
Apache Impala (incubating) integration

- CREATE TABLE ... DISTRIBUTION BY HASH(col1) INTO 16 BUCKETS
  AS SELECT ... FROM ...
- INSERT / UPDATE / DELETE

Remember MySQL with MyISAM?
Impala + Kudu “feels” just like that, but scale-out and optimized for analytics.

- Optimizations: predicate pushdown, scan locality, scan parallelism
- More optimizations on the way

Not an Impala user? Community working on other integrations (Apache Hive, Apache Drill, Presto, etc)
Apache Spark DataFrame/DataSource integration

// spark-shell --packages org.apache.kudu:kudu-spark_2.10:1.0.0
// Import kudu datasource
import org.kududb.spark.kudu._
val kuduDataFrame = sqlContext.read.options(
    Map("kudu.master" -> "master1,master2,master3",
        "kudu.table" -> "my_table_name"))

// Then query using Spark data frame API
kuduDataFrame.select("id").filter("id" >= 5).show()
// (prints the selection to the console)

// Or register kuduDataFrame as a table and use Spark SQL
kuduDataFrame.registerTempTable("my_table")
sqlContext.sql("select id from my_table where id >= 5").show()
// (prints the sql results to the console)
Writing from Spark

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

// 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 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

```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)]
```
Spark DataSource optimizations

Partition pruning

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

- Multi-framework cluster (MR + HDFS + Kudu on the same disks)
- KuduTableInputFormat / KuduTableOutputFormat
  - Support for pushing down predicates, column projections, etc.
  - Lots of Kudu integration / correctness testing done via MapReduce
Apache Flume integration

• Write streaming data to Kudu
• Specify a RegExp of your record format and the column name mapping. Flume will parse it for you and insert / upsert into Kudu.
• Then deploy with a Flume config file like the following:

agent.sink.kudu.type = org.kududb.flume.sink.KuduSink
agent.sink.kudu.masterAddresses = kudu01.example.com
agent.sink.kudu.tableName = my-table
agent.sink.kudu.producer = RegexpKuduOperationsProducer
agent.sink.kudu.producer.pattern = (?<key>\d+),(?<name>[w]+)

• Also supports Apache Avro serialized records
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
Performance improvements since 0.5.0

- Significant improvements to compaction throughput
- Default configurations tuned for much less write amplification
- Reduced lock contention on RPC system, block cache
- 2-3x improvement for selective filters on dictionary-encoded columns

Other speedups:
- **Startup time** 2-3x better (more work coming)
- **First scan** following restart ~2x faster
- More compact and compressed **internal index storage**
Performance improvements since 0.5.0 (YCSB)

Single node micro-benchmark, 500M record insert, 16 client threads, each record ~110 bytes
Runs Load, followed by ‘C’ (10min), followed by ‘A’ (10min)
Project status
Project status

• Apache Kudu is a top-level project (TLP) at the Apache Software Foundation
  • Community-driven open source process
• Kudu is considered to be stable and usable in production as of version 1.0.0
  • The latest release is version 1.0.1
  • Kudu 1.1.0 is up for release vote right now
• Useful for many applications (Xiaomi, JD are running in production)
  • Have never experienced unrecoverable data loss, reasonably stable (almost no crashes reported). Users operating up to 200 nodes so far
  • Still requires some expert assistance to run in production
What’s coming in Kudu 1.1.0, 1.2.0 and beyond

- **Security.** Kudu currently has no support for Kerberos security or access control. These features are critical for many use cases, and we’ll be working hard to add them.
- **Operability.** While operations have always been a focus of our development, as we move into more and more production workloads, the importance of having great tooling for operators only increases.
- **Performance.** Kudu is fast today, but we’ve got a whole roadmap ahead of us to make it even faster. This includes items like support for next-generation persistent memory hardware, as well as big gains in the performance of SQL engines on Kudu.
- **Scalability.** Kudu users are already running on 200 node clusters today, but we plan to continue working on stability and performance at scale.
Kudu Roadmap
Open Source “Roadmaps”?

• Kudu is an open source ASF project
• ASF governance means there is no guaranteed roadmap
  • Whatever people contribute is the roadmap!

• But I can speak to what I and my coworkers will be focusing on
• **Disclaimer:** quality-first mantra -> fuzzy timeline commitments
Security Roadmap

1) Kerberos **authentication**
   a) Client-server mutual authentication
   b) Server-server mutual authentication
   c) Execution framework-server authentication (delegation tokens)

2) Extra-coarse-grained **authorization**
   a) Likely a cluster-wide “allowed user list”

3) **Group/role mapping**
   a) LDAP/Unix/etc

4) **Data exposure hardening**
   a) e.g. ensure that web UIs don’t leak data

5) **Fine-grained authorization**
   a) Table/database/column level
Operability

1) Stability
   a) Continued stress testing, fault injection, etc
   b) Faster and safer recovery after failures

2) Recovery tools
   a) Repair from *minority* (eg if two hosts explode simultaneously)
   b) Replace from *empty* (eg if three hosts explode simultaneously)
   c) Repair file system state after power outage

3) Easier problem diagnosis
   a) Client “timeout” errors
   b) Easier performance issue diagnosis
Performance and scale

- **Read performance**
  - Dynamic predicates (aka runtime filters)
  - Spark statistics
  - Additional filter pushdown (e.g. “IN (...), “LIKE”)
  - I/O scheduling from spinning disks

- **Write performance**
  - Improved bulk load capability

- **Scalability**
  - Users planning to run 400 node clusters
  - Rack-aware placement
Client improvements roadmap

• Python
  • Full feature parity with C++ and Java
  • Even more pythonic
  • Integrations: Pandas, PySpark

• All clients:
  • More API documentation, tutorials, examples
  • Better error messages/exceptions
  • Full support for snapshot consistency level
Joining the growing community
Getting started as a developer

• Source code: github.com/apache/kudu - all commits go here first
• Code reviews: gerrit.cloudera.org - all code reviews are public
• Developer mailing list: dev@kudu.apache.org
• Public JIRA: issues.apache.org/jira/browse/KUDU - includes bug history since 2013

Contributions are welcome and strongly encouraged!
Getting started as a user

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

- 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