Apache Kudu User Guide

Cloudera, Inc.
1001 Page Mill Road, Bldg 3
Palo Alto, CA 94304
info@cloudera.com
US: 1-888-789-1488
Intl: 1-650-362-0488
www.cloudera.com

Release Information

Version: Apache Kudu 1.2.0 / CDH 5.10.x
Date: December 7, 2017

Table of Contents

About Apache Kudu

Apache Kudu is a columnar storage manager developed for the Hadoop platform. Kudu shares the common technical properties of Hadoop ecosystem applications: It runs on commodity hardware, is horizontally scalable, and supports highly available operation.

Apache Kudu is a top-level project in the Apache Software Foundation.

Kudu's benefits include:

- Fast processing of OLAP workloads.
- Integration with MapReduce, Spark, Flume, and other Hadoop ecosystem components.
- Tight integration with Apache Impala (incubating), making it a good, mutable alternative to using HDFS with Apache Parquet.
- Strong but flexible consistency model, allowing you to choose consistency requirements on a per-request basis, including the option for strict serialized consistency.
- Strong performance for running sequential and random workloads simultaneously.
- Easy administration and management through Cloudera Manager.
- High availability. Tablet Servers and Master use the Raft consensus algorithm, which ensures availability as long as more replicas are available than unavailable. Reads can be serviced by read-only follower tablets, even in the event of a leader tablet failure.
- Structured data model.

By combining all of these properties, Kudu targets support for applications that are difficult or impossible to implement on currently available Hadoop storage technologies. Applications for which Kudu is a viable solution include:

- Reporting applications where new data must be immediately available for end users
- Time-series applications that must support queries across large amounts of historic data while simultaneously returning granular queries about an individual entity
- Applications that use predictive models to make real-time decisions, with periodic refreshes of the predictive model based on all historical data

For more details, see Example Use Cases on page 11.

Kudu-Impala Integration Features

- **`CREATE/ALTER/DROP TABLE`** - Impala supports creating, altering, and dropping tables using Kudu as the persistence layer. The tables follow the same internal/external approach as other tables in Impala, allowing for flexible data ingestion and querying.
- **`INSERT`** - Data can be inserted into Kudu tables from Impala using the same mechanisms as any other table with HDFS or HBase persistence.
- **`UPDATE/DELETE`** - Impala supports the `UPDATE` and `DELETE` SQL commands to modify existing data in a Kudu table row-by-row or as a batch. The syntax of the SQL commands is chosen to be as compatible as possible with existing solutions. In addition to simple `DELETE` or `UPDATE` commands, you can specify complex joins in the `FROM` clause of the query, using the same syntax as a regular `SELECT` statement.
- **Flexible Partitioning** - Similar to partitioning of tables in Hive, Kudu allows you to dynamically pre-split tables by hash or range into a predefined number of tablets, in order to distribute writes and queries evenly across your cluster. You can partition by any number of primary key columns, with any number of hashes, a list of split rows, or a combination. A partition scheme is required.
- **Parallel Scan** - To achieve the highest possible performance on modern hardware, the Kudu client used by Impala parallelizes scans across multiple tablets.
- **High-efficiency queries** - Where possible, Impala pushes down predicate evaluation to Kudu, so that predicates are evaluated as close as possible to the data. Query performance is comparable to Parquet in many workloads.

Concepts and Terms

Columnar Datastore

Kudu is a *columnar datastore*. A columnar datastore stores data in strongly-typed columns. With a proper design, a columnar store can be superior for analytical or data warehousing workloads for several reasons.

Read Efficiency

For analytical queries, you can read a single column, or a portion of that column, while ignoring other columns. This means you can fulfill your request while reading a minimal number of blocks on disk. With a row-based store, you need to read the entire row, even if you only return values from a few columns.

Data Compression

Because a given column contains only one type of data, pattern-based compression can be orders of magnitude more efficient than compressing mixed data types, which are used in row-based solutions. Combined with the efficiencies of reading data from columns, compression allows you to fulfill your query while reading even fewer blocks from disk.

Raft Consensus Algorithm

The Raft consensus algorithm provides a way to elect a *leader* for a distributed cluster from a pool of potential leaders. If a follower cannot reach the current leader, it transitions itself to become a *candidate*. Given a quorum of voters, one candidate is elected to be the new leader, and the others transition back to being followers. A full discussion of Raft is out of scope for this documentation, but it is a robust algorithm.

Kudu uses the Raft Consensus Algorithm for the election of masters and leader tablets, as well as determining the success or failure of a given write operation.

Table

A *table* is where your data is stored in Kudu. A table has a schema and a totally ordered primary key. A table is split into segments called tablets, by primary key.

Tablet

A *tablet* is a contiguous segment of a table, similar to a *partition* in other data storage engines or relational databases. A given tablet is replicated on multiple tablet servers, and at any given point in time, one of these replicas is considered the leader tablet. Any replica can service reads, and writes require consensus among the set of tablet servers serving the tablet.

Tablet Server

A *tablet server* stores and serves tablets to clients. For a given tablet, one tablet server acts as a leader and the others serve follower replicas of that tablet. Only leaders service write requests, while leaders or followers each service read requests. Leaders are elected using Raft consensus. One tablet server can serve multiple tablets, and one tablet can be served by multiple tablet servers.

Master

The *master* keeps track of all the tablets, tablet servers, the catalog table, and other metadata related to the cluster. At a given point in time, there can only be one acting master (the leader). If the current leader disappears, a new master is elected using Raft consensus.

The master also coordinates metadata operations for clients. For example, when creating a new table, the client internally sends the request to the master. The master writes the metadata for the new table into the catalog table, and coordinates the process of creating tablets on the tablet servers.

All the master's data is stored in a tablet, which can be replicated to all the other candidate masters.

Tablet servers heartbeat to the master at a set interval (the default is once per second).

Catalog Table

The *catalog table* is the central location for metadata of Kudu. It stores information about tables and tablets. The catalog table is accessible to clients through the master, using the client API. The catalog table may not be read or written directly. Instead, it is accessible only via metadata operations exposed in the client API. The catalog table stores two categories of metadata:

Contents of the Catalog Table	
Tables	table schemas, locations, and states
Tablets	the list of existing tablets, which tablet servers have replicas of each tablet, the tablet's current state, and start and end keys.

Logical Replication

Kudu replicates operations, not on-disk data. This is referred to as *logical replication*, as opposed to *physical replication*. This has several advantages:

- Although inserts and updates do transmit data over the network, deletes do not need to move any data. The delete operation is sent to each tablet server, which performs the delete locally.
- Physical operations, such as compaction, do not need to transmit the data over the network in Kudu. This is different from storage systems that use HDFS, where the blocks need to be transmitted over the network to fulfill the required number of replicas.
- Tablets do not need to perform compactions at the same time or on the same schedule, or otherwise remain in sync on the physical storage layer. This decreases the chances of all tablet servers experiencing high latency at the same time, due to compactions or heavy write loads.

Architectural Overview

The following diagram shows a Kudu cluster with three masters and multiple tablet servers, each serving multiple tablets. It illustrates how Raft consensus is used to allow for both leaders and followers for both the masters and tablet servers. In addition, a tablet server can be a leader for some tablets, and a follower for others. Leaders are shown in gold, while followers are shown in blue.

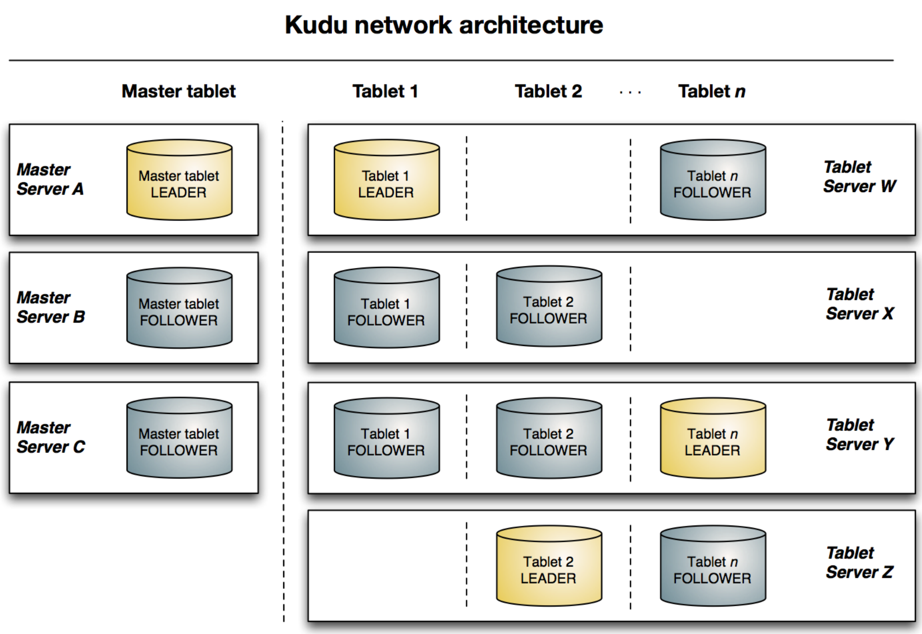


Figure 1: Kudu Architectural Overview

Example Use Cases

Streaming Input with Near Real Time Availability

A common business challenge is one where new data arrives rapidly and constantly, and the same data needs to be available in near real time for reads, scans, and updates. Kudu offers the powerful combination of fast inserts and updates with efficient columnar scans to enable real-time analytics use cases on a single storage layer.

Time-Series Application with Widely Varying Access Patterns

A time-series schema is one in which data points are organized and keyed according to the time at which they occurred. This can be useful for investigating the performance of metrics over time or attempting to predict future behavior based on past data. For instance, time-series customer data might be used both to store purchase click-stream history and to predict future purchases, or for use by a customer support representative. While these different types of analysis are occurring, inserts and mutations may also be occurring individually and in bulk, and become available immediately to read workloads. Kudu can handle all of these access patterns simultaneously in a scalable and efficient manner.

Kudu is a good fit for time-series workloads for several reasons. With Kudu's support for hash-based partitioning, combined with its native support for compound row keys, it is simple to set up a table spread across many servers without the risk of "hotspotting" that is commonly observed when range partitioning is used. Kudu's columnar storage engine is also beneficial in this context, because many time-series workloads read only a few columns, as opposed to the whole row.

In the past, you might have needed to use multiple datastores to handle different data access patterns. This practice adds complexity to your application and operations, and duplicates your data, doubling (or worse) the amount of storage required. Kudu can handle all of these access patterns natively and efficiently, without the need to off-load work to other datastores.

Predictive Modeling

Data scientists often develop predictive learning models from large sets of data. The model and the data may need to be updated or modified often as the learning takes place or as the situation being modeled changes. In addition, the scientist may want to change one or more factors in the model to see what happens over time. Updating a large set of data stored in files in HDFS is resource-intensive, as each file needs to be completely rewritten. In Kudu, updates happen in near real time. The scientist can tweak the value, re-run the query, and refresh the graph in seconds or minutes, rather than hours or days. In addition, batch or incremental algorithms can be run across the data at any time, with near-real-time results.

Combining Data In Kudu With Legacy Systems

Companies generate data from multiple sources and store it in a variety of systems and formats. For instance, some of your data may be stored in Kudu, some in a traditional RDBMS, and some in files in HDFS. You can access and query all of these sources and formats using Impala, without the need to change your legacy systems.

Next Steps

- Read about installing Kudu.
- Learn about using Kudu with Impala.

Apache Kudu Release Notes

This topic includes the release notes for all beta and generally available versions (GA) of Apache Kudu.

Schema Design and Usage Limitations

The Apache Kudu Schema Design and Usage Limitations on page 32 topic describes known issues and limitations with respect to schema design, integration with Apache Impala, and security in Kudu, as of the current release.

Kudu 1.2.0 / CDH 5.10.2 Release Notes

Apache Kudu 1.2.x / CDH 5.10.2 includes the following fixed issues.

- KUDU-1933 - Fixed an issue that truncated the 64-bit log index in the OpId to 32 bits, causing overflow of the log index.
- KUDU-1607 - Fixed an issue where Kudu could not delete failed tablets using the `DROP TABLE` command.
- KUDU-1905 - Allow reinserts on tables when all columns are part of the primary key.
- KUDU-1893 - Made a fix to avoid incorrect NULL results and ensure evaluation of predicates for columns added after table creation.

Kudu 1.2.0 / CDH 5.10.1 Release Notes

Apache Kudu 1.2.x / CDH 5.10.1 includes the following fixed issues.

- KUDU-1904 - Fixed a bug where RLE columns with only NULL values would crash on scan.
- KUDU-1899 - Fixed an issue where tablet servers would crash after inserting an empty string primary key (`""`).
- KUDU-1851 - Fixed an issue with the Python client which would crash whenever a TableAlterer is instantiated directly.
- KUDU-1852 - `KuduTableAlterer` will no longer crash when given `nullptr` range bound arguments.
- KUDU-1821 - Improved warnings when the catalog manager starts.

Kudu 1.2.0 / CDH 5.10.0 Release Notes

> **Note:** Apache Kudu 1.2.0 / CDH 5.10.0 release is the first Kudu release to be supported with Cloudera Manager 5.10.0 and CDH 5.10.0. Any future Kudu 1.2.0 maintenance releases will likely be supported by corresponding CDH and Cloudera Manager 5.10.x maintenance releases.

Apache Kudu 1.2.x / CDH 5.10.0 includes the following new features, fixed issues, and changes.

New Features and Improvements in Kudu 1.2.0 / CDH 5.10.0

See also Issues resolved for Kudu 1.2.0 and Git changes between 1.1.x and 1.2.x.

New Features

- Kudu clients and servers now redact user data such as cell values from log messages, Java exception messages, and Status strings. User metadata such as table names, column names, and partition bounds are not redacted.
- Kudu's ability to provide consistency guarantees has been substantially improved:
  - Replicas now correctly track their "safe timestamp". This timestamp is the maximum timestamp at which reads are guaranteed to be repeatable.

  - A scan created using the SCAN_AT_SNAPSHOT mode will now either wait for the requested snapshot to be "safe" at the replica being scanned, or be re-routed to a replica where the requested snapshot is "safe". This ensures that all such scans are repeatable.
  - Kudu Tablet Servers now properly retain historical data when a row with a given primary key is inserted and deleted, followed by the insertion of a new row with the same key. Previous versions of Kudu would not retain history in such situations. This allows the server to return correct results for snapshot scans with a timestamp in the past, even in the presence of such "reinsertion" scenarios.
  - The Kudu clients now automatically retain the timestamp of their latest successful read or write operation. Scans using the READ_AT_SNAPSHOT mode without a client-provided timestamp automatically assign a timestamp higher than the timestamp of their most recent write. Writes also propagate the timestamp, ensuring that sequences of operations with causal dependencies between them are assigned increasing timestamps. Together, these changes allow clients to achieve read-your-writes consistency, and also ensure that snapshot scans performed by other clients return causally-consistent results.

- User data in log files is now redacted by default.
- Kudu servers now automatically limit the number of log files being stored. By default, 10 log files will be retained at each severity level.

Optimizations and Improvements

- The logging in the Java and cpp clients has been substantially quieted. Clients no longer log messages in normal operation unless there is some kind of error.
- The cpp client now includes a KuduSession::SetErrorBufferSpace API which can limit the amount of memory used to buffer errors from asynchronous operations.
- The Java client now fetches tablet locations from the Kudu Master in batches of 1000, increased from batches of 10 in prior versions. This can substantially improve the performance of Spark and Impala queries running against Kudu tables with large numbers of tablets.
- Table metadata lock contention in the Kudu Master was substantially reduced. This improves the performance of tablet location lookups on large clusters with a high degree of concurrency.
- Lock contention in the Kudu Tablet Server during high-concurrency write workloads was also reduced. This can reduce CPU consumption and improve performance when a large number of concurrent clients are writing to a smaller number of a servers.
- Lock contention when writing log messages has been substantially reduced. This source of contention could cause high tail latencies on requests, and when under high load could contribute to cluster instability such as election storms and request timeouts.
- The BITSHUFFLE column encoding has been optimized to use the AVX2 instruction set present on processors including Intel(R) Sandy Bridge and later. Scans on BITSHUFFLE-encoded columns are now up to 30% faster.
- The kudu tool now accepts hyphens as an alternative to underscores when specifying actions. For example, kudu local-replica copy-from-remote may be used as an alternative to kudu local_replica copy_from_remote.

Issues Fixed in Kudu 1.2.0 / CDH 5.10.0

See Issues resolved for Kudu 1.2.0 and Git changes between 1.1.x and 1.2.x.

- KUDU-1508 - Fixed a long-standing issue in which running Kudu on ext4 file systems could cause file system corruption. While this issue has been known to still manifest in certain rare cases, the corruption is harmless and can be repaired as part of a regular fsck. Switching from ext4 to xfs will also solve the problem.
- KUDU-1399 - Implemented an LRU cache for open files, which prevents running out of file descriptors on long-lived Kudu clusters. By default, Kudu will limit its file descriptor usage to half of its configured ulimit.
- Gerrit #5192 - Fixed an issue which caused data corruption and crashes in the case that a table had a non-composite (single-column) primary key, and that column was specified to use DICT_ENCODING or BITSHUFFLE encodings. If a table with an affected schema was written in previous versions of Kudu, the corruption will not be automatically repaired; users are encouraged to re-insert such tables after upgrading to Kudu 1.2 or later.
- Gerrit #5541 - Fixed a bug in the Spark KuduRDD implementation which could cause rows in the result set to be silently skipped in some cases.

- KUDU-1551 - Fixed an issue in which the tablet server would crash on restart in the case that it had previously crashed during the process of allocating a new WAL segment.
- KUDU-1764 - Fixed an issue where Kudu servers would leak approximately 16-32MB of disk space for every 10GB of data written to disk. After upgrading to Kudu 1.2 or later, any disk space leaked in previous versions will be automatically recovered on startup.
- KUDU-1750 - Fixed an issue where the API to drop a range partition would drop any partition with a matching lower _or_ upper bound, rather than any partition with matching lower _and_ upper bound.
- KUDU-1766 - Fixed an issue in the Java client where equality predicates which compared an integer column to its maximum possible value (e.g. `Integer.MAX_VALUE`) would return incorrect results.
- KUDU-1780 - Fixed the kudu-client Java artifact to properly shade classes in the `com.google.thirdparty` namespace. The lack of proper shading in prior releases could cause conflicts with certain versions of Google Guava.
- Gerrit #5327 - Fixed shading issues in the `kudu-flume-sink` Java artifact. The sink now expects that Hadoop dependencies are provided by Flume, and properly shades the Kudu client's dependencies.
- Fixed a few issues using the Python client library from Python 3.

Incompatible Changes in Kudu 1.2.0 / CDH 5.10.0

Apache Kudu 1.2.0 introduces the following incompatible changes:

- The replication factor of tables is now limited to a maximum of 7. In addition, it is no longer allowed to create a table with an even replication factor.
- The `GROUP_VARINT` encoding is now deprecated. Kudu servers have never supported this encoding, and now the client-side constant has been deprecated to match the server's capabilities.
- **Client Library Compatibility**
  - The Kudu 1.2 Java client is API- and ABI-compatible with Kudu 1.1. Applications written against Kudu 1.1 will compile and run against the Kudu 1.2 client and vice-versa.
  - The Kudu 1.2 cpp client is API- and ABI-forward-compatible with Kudu 1.1. Applications written and compiled against the Kudu 1.1 client will run without modification against the Kudu 1.2 client. Applications written and compiled against the Kudu 1.2 client will run without modification against the Kudu 1.1 client unless they use one of the following new APIs:
    - `kudu::DisableSaslInitialization()`
    - `KuduSession::SetErrorBufferSpace(...)`
  - The Kudu 1.2 Python client is API-compatible with Kudu 1.1. Applications written against Kudu 1.1 will continue to run against the Kudu 1.2 client and vice-versa.

Known Issues and Limitations in Kudu 1.2.0 / CDH 5.10.0

- KUDU-1893 - Certain query results incorrectly include rows with null values for predicates.

  Solution - Upgrade to the latest maintenance release.

- **Schema and Usage Limitations** - For a complete list of schema design and usage limitations for Apache Kudu, see Apache Kudu Schema Design and Usage Limitations on page 32.

Kudu 1.1.x Release Notes

Apache Kudu 1.1 includes the following new features and fixed issues.

New Features in Kudu 1.1.0

See also Issues resolved for Kudu 1.1.0 and Git changes between 1.0.x and 1.1.x.

- The Python client has been brought up to feature parity with the Java and C++ clients and as such the package version will be brought to 1.1 with this release (from 0.3). A list of the highlights can be found below.
  - Improved Partial Row semantics
  - Range partition support
  - Scan Token API
  - Enhanced predicate support
  - Support for all Kudu data types (including a mapping of Python's `datetime.datetime` to `UNIXTIME_MICROS`)
  - Alter table support
  - Enabled Read at Snapshot for Scanners
  - Enabled Scanner Replica Selection
  - A few bug fixes for Python 3 in addition to various other improvements.
- `IN LIST` predicate pushdown support was added to allow optimized execution of filters which match on a set of column values. Support for Spark, Map Reduce and Impala queries utilizing `IN LIST` pushdown is not yet complete.
- The Java client now features client-side request tracing in order to help troubleshoot timeouts. Error messages are now augmented with traces that show which servers were contacted before the timeout occurred instead of just the last error. The traces also contain RPCs that were required to fulfill the client's request, such as contacting the master to discover a tablet's location. Note that the traces are not available for successful requests and cannot be queried.

Performance

- Kudu now publishes JAR files for Spark 2.0 compiled with Scala 2.11 along with the existing Spark 1.6 JAR compiled with Scala 2.10.
- The Java client now allows configuring scanners to read from the closest replica instead of the known leader replica. The default remains the latter. Use the relevant `ReplicaSelection` enum with the scanner's builder to change this behavior.

Wire protocol compatibility

- The Java client's sync API (`KuduClient`, `KuduSession`, `KuduScanner`) used to throw either a `NonRecoverableException` or a `TimeoutException` for a timeout, and now it's only possible for the client to throw the former.
- The Java client's handling of errors in `KuduSession` was modified so that subclasses of `KuduException` are converted into RowErrors instead of being thrown.

Command line tools

- The tool `kudu tablet leader_step_down` has been added to manually force a leader to step down.
- The tool `kudu remote_replica copy` has been added to manually copy a replica from one running tablet server to another.
- The tool `kudu local_replica delete` has been added to delete a replica of a tablet.
- The `kudu test loadgen` tool has been added to replace the obsoleted `insert-generated-rows` standalone binary. The new tool is enriched with additional functionality and can be used to run load generation tests against a Kudu cluster.

Client APIs (C++/Java/Python)

- The C++ client no longer requires the old gcc5 ABI. Which ABI is actually used depends on the compiler configuration. Some new distros (e.g. Ubuntu 16.04) will use the new ABI. Your application must use the same ABI as is used by the client library; an easy way to guarantee this is to use the same compiler to build both.
- The C++ client's `KuduSession::CountBufferedOperations()` method is deprecated. Its behavior is inconsistent unless the session runs in the `MANUAL_FLUSH` mode. Instead, to get number of buffered operations, count invocations of the `KuduSession::Apply()` method since last `KuduSession::Flush()` call or, if using asynchronous flushing, since last invocation of the callback passed into `KuduSession::FlushAsync()`.
- The Java client's `OperationResponse.getWriteTimestamp` method was renamed to `getWriteTimestampRaw` to emphasize that it doesn't return milliseconds, unlike what its Javadoc indicated. The renamed method was also hidden from the public APIs and should not be used.
- The Java client's sync API (`KuduClient`, `KuduSession`, `KuduScanner`) used to throw either a `NonRecoverableException` or a `TimeoutException` for a timeout, and now it's only possible for the client to throw the former.
- The Java client's handling of errors in `KuduSession` was modified so that subclasses of `KuduException` are converted into RowErrors instead of being thrown.

Issues Fixed in Kudu 1.1.0

See Issues resolved for Kudu 1.1.0 and Git changes between 1.0.x and 1.1.x.

Kudu 1.0.1 Release Notes

Apache Kudu 1.0.1 is a bug fix release, with no new features or backwards incompatible changes.

Issues Fixed in Kudu 1.0.1

- KUDU-1681: Fixed a bug in the tablet server which could cause a crash when the DNS lookup during master heartbeat failed.
- KUDU-1660: Fixed a bug which would cause the Kudu master and tablet server to fail to start on single CPU systems.
- KUDU-1652: Fixed a bug that would cause the C++ client, tablet server, and Java client to crash or throw an exception when attempting to scan a table with a predicate which simplifies to `IS NOT NULL` on a non-nullable column. For example, setting a '`<= 127`' predicate on an `INT8` column could trigger this bug, since the predicate only filters null values.
- KUDU-1651: Fixed a bug that would cause the tablet server to crash when evaluating a scan with predicates over a dictionary encoded column containing an entire block of null values.
- KUDU-1623: Fixed a bug that would cause the tablet server to crash when handling UPSERT operations that only set values for the primary key columns.
- Gerrit #4488: Fixed a bug in the Java client's `KuduException` class which could cause an unexpected `NullPointerException` to be thrown when the exception did not have an associated message.
- KUDU-1090: Fixed a bug in the memory tracker which could cause a rare crash during tablet server startup.

Kudu 1.0.0 Release Notes

After approximately a year of beta releases, Apache Kudu has reached version 1.0. This version number signifies that the development team feels that Kudu is stable enough for usage in production environments.

Kudu 1.0.0 delivers a number of new features, bug fixes, and optimizations.

To upgrade Kudu to 1.0.0, see Upgrade Parcels or Upgrade Packages.

Other Noteworthy Changes

- This is the first non-beta release of the Apache Kudu project. (Although because Kudu is not currently integrated into CDH, it is not yet an officially supported CDH component.)

New Features in Kudu 1.0.0

See also Issues resolved for Kudu 1.0.0 and Git changes between 0.10.0 and 1.0.0.

- Removal of multiversion concurrency control (MVCC) history is now supported. This is known as tablet history GC. This allows Kudu to reclaim disk space, where previously Kudu would keep a full history of all changes made to a given table since the beginning of time. Previously, the only way to reclaim disk space was to drop a table.
- Kudu will still keep historical data, and the amount of history retained is controlled by setting the configuration flag `--tablet_history_max_age_sec`, which defaults to 15 minutes (expressed in seconds). The timestamp represented by the current time minus `tablet_history_max_age_sec` is known as the ancient history mark (AHM). When a compaction or flush occurs, Kudu will remove the history of changes made prior to the ancient history mark. This only affects historical data; currently-visible data will not be removed. A specialized maintenance manager background task to remove existing "cold" historical data that is not in a row affected by the normal compaction process will be added in a future release.
- Most of Kudu's command line tools have been consolidated under a new top-level `kudu` tool. This reduces the number of large binaries distributed with Kudu and also includes much-improved help output.
- The Kudu Flume Sink now supports processing events containing Avro-encoded records, using the new `AvroKuduOperationsProducer`.
- Administrative tools including `kudu cluster ksck` now support running against multi-master Kudu clusters.
- The output of the `ksck` tool is now colorized and much easier to read.
- The C++ client API now supports writing data in `AUTO_FLUSH_BACKGROUND` mode. This can provide higher throughput for ingest workloads.

Performance

- The performance of comparison predicates on dictionary-encoded columns has been substantially optimized. Users are encouraged to use dictionary encoding on any string or binary columns with low cardinality, especially if these columns will be filtered with predicates.
- The Java client is now able to prune partitions from scanners based on the provided predicates. For example, an equality predicate on a hash-partitioned column will now only access those tablets that could possibly contain matching data. This is expected to improve performance for the Spark integration as well as applications using the Java client API.
- The performance of compaction selection in the tablet server has been substantially improved. This can increase the efficiency of the background maintenance threads and improve overall throughput of heavy write workloads.
- The policy by which the tablet server retains write-ahead log (WAL) files has been improved so that it takes into account other replicas of the tablet. This should help mitigate the spurious eviction of tablet replicas on machines that temporarily lag behind the other replicas.

Wire protocol compatibility

- Kudu 1.0.0 maintains client-server wire-compatibility with previous releases. Applications using the Kudu client libraries may be upgraded either before, at the same time, or after the Kudu servers.
- Kudu 1.0.0 does *not* maintain server-server wire compatibility with previous releases. Therefore, rolling upgrades between earlier versions of Kudu and Kudu 1.0.0 are not supported.

Incompatible Changes in Kudu 1.0.0

Command line tools

- The `kudu-pbc-dump` tool has been removed. The same functionality is now implemented as `kudu pbc dump`.
- The `kudu-ksck` tool has been removed. The same functionality is now implemented as `kudu cluster ksck`.
- The `cfile-dump` tool has been removed. The same functionality is now implemented as `kudu fs cfile dump`.
- The `log-dump` tool has been removed. The same functionality is now implemented as `kudu wal dump` and `kudu local_replica dump wals`.
- The `kudu-admin` tool has been removed. The same functionality is now implemented within `kudu table` and `kudu tablet`.
- The `kudu-fs_dump` tool has been removed. The same functionality is now implemented as `kudu fs dump`.
- The `kudu-ts-cli` tool has been removed. The same functionality is now implemented within `kudu master`, `kudu remote_replica`, and `kudu tserver`.
- The `kudu-fs_list` tool has been removed and some similar useful functionality has been moved under `kudu local_replica`.

Configuration flags

Some configuration flags are marked 'unsafe' and 'experimental'. Such flags are disabled by default. You may access these flags by enabling the additional flags, `--unlock_unsafe_flags` and `--unlock_experimental_flags`. Note that these flags might be removed or modified without a deprecation period or any prior notice in future Kudu releases. Cloudera does not support using unsafe and experimental flags. As a rule of thumb, Cloudera will not support any configuration flags not explicitly documented in the Kudu Configuration Reference Guide.

Client APIs (C++/Java/Python)

The `TIMESTAMP` column type has been renamed to `UNIXTIME_MICROS` in order to reduce confusion between Kudu's timestamp support and the timestamps supported by other systems such as Apache Hive and Apache Impala (incubating). Existing tables will automatically be updated to use the new name for the type.

Clients upgrading to the new client libraries must move to the new name for the type. Clients using old client libraries will continue to operate using the old type name, even when connected to clusters that have been upgraded. Similarly, if clients are upgraded before servers, existing timestamp columns will be available using the new type name.

`KuduSession` methods in the C++ library are no longer advertised as thread-safe to have one set of semantics for both C++ and Java Kudu client libraries.

The `KuduScanToken::TabletServers` method in the C++ library has been removed. The same information can now be found in the `KuduScanToken::tablet` method.

Apache Flume Integration

The `KuduEventProducer` interface used to process Flume events into Kudu operations for the Kudu Flume Sink has changed, and has been renamed `KuduOperationsProducer`. The existing KuduEventProducers have been updated for the new interface, and have been renamed similarly.

Known Issues and Limitations of Kudu 1.0.0

Schema and Usage Limitations

- Kudu is primarily designed for analytic use cases. You are likely to encounter issues if a single row contains multiple kilobytes of data.
- The columns which make up the primary key must be listed first in the schema.
- Key columns cannot be altered. You must drop and recreate a table to change its keys.

- Key columns must not be null.
- Columns with `DOUBLE`, `FLOAT`, or `BOOL` types are not allowed as part of a primary key definition.
- Type and nullability of existing columns cannot be changed by altering the table.
- A table's primary key cannot be changed.
- Dropping a column does not immediately reclaim space. Compaction must run first. There is no way to run compaction manually, but dropping the table will reclaim the space immediately.

Partitioning Limitations

- Tables must be manually pre-split into tablets using simple or compound primary keys. Automatic splitting is not yet possible. Range partitions may be added or dropped after a table has been created. See Schema Design for more information.
- Data in existing tables cannot currently be automatically repartitioned. As a workaround, create a new table with the new partitioning and insert the contents of the old table.

Replication and Backup Limitations

- Kudu does not currently include any built-in features for backup and restore. Users are encouraged to use tools such as Spark or Impala to export or import tables as necessary.

Impala Limitations

- To use Kudu with Impala, you must install a special release of Impala called Impala_Kudu. Obtaining and installing a compatible Impala release is detailed in Using Apache Impala (incubating) with Kudu on page 56.
- To use Impala_Kudu alongside an existing Impala instance, you must install using parcels.
- Updates, inserts, and deletes via Impala are non-transactional. If a query fails part of the way through, its partial effects will not be rolled back.
- All queries will be distributed across all Impala hosts which host a replica of the target table(s), even if a predicate on a primary key could correctly restrict the query to a single tablet. This limits the maximum concurrency of short queries made via Impala.
- No `TIMESTAMP` and `DECIMAL` type support. (The underlying Kudu type formerly known as `TIMESTAMP` has been renamed to `UNIXTIME_MICROS`; currently, there is no Impala-compatible `TIMESTAMP` type.)
- The maximum parallelism of a single query is limited to the number of tablets in a table. For good analytic performance, aim for 10 or more tablets per host or use large tables.
- Impala is only able to push down predicates involving =, <=, >=, or `BETWEEN` comparisons between any column and a literal value, and < and > for integer columns only. For example, for a table with an integer key `ts`, and a string key `name`, the predicate `WHERE ts >= 12345` will convert into an efficient range scan, whereas `where name > 'lipcon'` will currently fetch all data from the table and evaluate the predicate within Impala.

Security Limitations

- Authentication and authorization features are not implemented.
- Data encryption is not built in. Kudu has been reported to run correctly on systems using local block device encryption (e.g. dmcrypt).

Client and API Limitations

- `ALTER TABLE` is not yet fully supported via the client APIs. More `ALTER TABLE` operations will become available in future releases.

Other Known Issues

The following are known bugs and issues with the current release of Kudu. They will be addressed in later releases. Note that this list is not exhaustive, and is meant to communicate only the most important known issues.

- If the Kudu master is configured with the `-log_fsync_all` option, tablet servers and clients will experience frequent timeouts, and the cluster may become unusable.
- If a tablet server has a very large number of tablets, it may take several minutes to start up. It is recommended to limit the number of tablets per server to 100 or fewer. Consider this limitation when pre-splitting your tables. If you notice slow start-up times, you can monitor the number of tablets per server in the web UI.
- Due to a known bug in Linux kernels prior to 3.8, running Kudu on ext4 mount points may cause a subsequent fsck to fail with errors such as `Logical start <N> does not match logical start <M> at next level` . These errors are repairable using `fsck -y`, but may impact server restart time.

  This affects RHEL/CentOS 6.8 and below. A fix is planned for RHEL/CentOS 6.9. RHEL 7.0 and higher are not affected. Ubuntu 14.04 and later are not affected. SLES 12 and later are not affected.

Issues Fixed in Kudu 1.0.0

See Issues resolved for Kudu 1.0.0 and Git changes between 0.10.0 and 1.0.0.

Kudu 0.10.0 Release Notes

Kudu 0.10.0 delivers a number of new features, bug fixes, and optimizations.

See also Issues resolved for Kudu 0.10.0 and Git changes between 0.9.1 and 0.10.0.

To upgrade Kudu to 0.10.0, see Upgrade Parcels or Upgrade Packages.

Kudu 0.10.0 maintains wire-compatibility with previous releases, meaning that applications using the Kudu client libraries may be upgraded either before, at the same time, or after the Kudu servers. However, if you begin using new features of Kudu 0.10.0 such as manually range-partitioned tables, you must first upgrade all clients to this release.

After upgrading to Kudu 0.10.0, it is possible to downgrade to 0.9.x with the following exceptions:

- Tables created in 0.10.0 will not be accessible after a downgrade to 0.9.x.
- A multi-master setup formatted in 0.10.0 may not be downgraded to 0.9.x.

This release does not maintain full Java API or ABI compatibility with Kudu 0.9.x due to a package rename and some other small changes. See Incompatible Changes in Kudu 0.10.0 on page 22 for details.

Other Noteworthy Changes

- This is the first release of Apache Kudu as a top-level (non-incubating) project.
- The default false positive rate for Bloom filters has been changed from 1% to 0.01%. This will increase the space consumption of Bloom filters by a factor of two (from approximately 10 bits per row to approximately 20 bits per row). This is expected to substantially improve the performance of random-write workloads at the cost of an incremental increase in disk space usage.
- The Kudu C++ client library now has Doxygen-based API documentation available online.
- Kudu now uses the Raft consensus algorithm even for unreplicated tables. This change simplifies code and will also allow administrators to enable replication on a previously unreplicated table. This change is internal and should not be visible to users.

New Features in Kudu 0.10.0

- Users may now manually manage the partitioning of a range-partitioned table. When a table is created, the user may specify a set of range partitions that do not cover the entire available key space. A user may add or drop range partitions to existing tables.

This feature can be particularly helpful with time series workloads in which new partitions can be created on an hourly or daily basis. Old partitions may be efficiently dropped if the application does not need to retain historical data past a certain point.

- Support for running Kudu clusters with multiple masters has been stabilized. Users may start a cluster with three or five masters to provide fault tolerance despite a failure of one or two masters, respectively.

  Certain tools such as `ksck` lack complete support for multiple masters. These deficiencies will be addressed in a following release.

- Kudu now supports the ability to reserve a certain amount of free disk space in each of its configured data directories. If a directory's free disk space drops to less than the configured minimum, Kudu will stop writing to that directory until space becomes available. If no space is available in any configured directory, Kudu will abort.

  This feature may be configured using the `--fs_data_dirs_reserved_bytes` and `--fs_wal_dir_reserved_bytes` flags.

- The Spark integration's `KuduContext` now supports four new methods for writing to Kudu tables: `insertRows`, `upsertRows`, `updateRows`, and `deleteRows`. These are now the preferred way to write to Kudu tables from Spark.

Other Improvements in Kudu 0.10.0

- KUDU-1516: The `kudu-ksck` tool has been improved and now detects problems such as when a tablet does not have a majority of replicas on live tablet servers, or if those replicas aren't in a good state. Users who currently depend on the tool to detect inconsistencies may now see failures when before they wouldn't see any.

- Gerrit #3477: The way operations are buffered in the Java client has been reworked. Previously, the session's buffer size was set per tablet, meaning that a buffer size of 1,000 for 10 tablets being written to allowed for 10,000 operations to be buffered at the same time. With this change, all the tablets share one buffer, so users might need to set a bigger buffer size in order to reach the same level of performance as before.

- Gerrit #3674: Added `LESS` and `GREATER` options for column predicates.

- KUDU-1444: Added support for passing back basic per-scan metrics, such as cache hit rate, from the server to the C++ client. See the `KuduScanner::GetResourceMetrics()` API for detailed usage. This feature will be supported in the Java client API in a future release.

- KUDU-1446: Improved the order in which the tablet server evaluates predicates, so that predicates on smaller columns are evaluated first. This may improve performance on queries which apply predicates on multiple columns of different sizes.

- KUDU-1398: Improved the storage efficiency of Kudu's internal primary key indexes. This optimization should decrease space usage and improve random access performance, particularly for workloads with lengthy primary keys.

Issues Fixed in Kudu 0.10.0

- Gerrit #3541: Fixed a problem in the Java client whereby an RPC could be dropped when a connection to a tablet server or master was forcefully closed on the server-side while RPCs to that server were in the process of being encoded. The effect was that the RPC would not be sent, and users of the synchronous API would receive a `TimeoutException`. Several other Java client bugs which could cause similar spurious timeouts were also fixed in this release.
- Gerrit #3724: Fixed a problem in the Java client whereby an RPC could be dropped when a socket timeout was fired while that RPC was being sent to a tablet server or master. This would manifest itself in the same way as Gerrit #3541.
- KUDU-1538: Fixed a bug in which recycled block identifiers could cause the tablet server to lose data. Following this bug fix, block identifiers will no longer be reused.

Incompatible Changes in Kudu 0.10.0

- Gerrit #3737: The Java client has been repackaged under `org.apache.kudu` instead of `org.kududb`. Import statements for Kudu classes must be modified in order to compile against 0.10.0. Wire compatibility is maintained.
- Gerrit #3055: The Java client's synchronous API methods now throw `KuduException` instead of `Exception`. Existing code that catches `Exception` should still compile, but introspection of an exception's message may be impacted. This change was made to allow thrown exceptions to be queried more easily using `KuduException.getStatus` and calling one of `Status`'s methods. For example, an operation that tries to delete a table that doesn't exist would return a `Status` that returns true when queried on `isNotFound()`.
- The Java client's `KuduTable.getTabletsLocations` set of methods is now deprecated. Additionally, they now take an exclusive end partition key instead of an inclusive key. Applications are encouraged to use the scan tokens API instead of these methods in the future.
- The C++ API for specifying split points on range-partitioned tables has been improved to make it easier for callers to properly manage the ownership of the provided rows.
- The `TableCreator::split_rows` API took a `vector<const KuduPartialRow*>`, which made it very difficult for the calling application to do proper error handling with cleanup when setting the fields of the `KuduPartialRow`. This API has been now deprecated and replaced by a new method `TableCreator::add_range_split` which allows easier use of smart pointers for safe memory management.
- The Java client's internal buffering has been reworked. Previously, the number of buffered write operations was constrained on a per-tablet-server basis. Now, the configured maximum buffer size constrains the total number of buffered operations across all tablet servers in the cluster. This provides a more consistent bound on the memory usage of the client regardless of the size of the cluster to which it is writing. This change can negatively affect the write performance of Java clients which rely on buffered writes. Consider using the `setMutationBufferSpace` API to increase a session's maximum buffer size if write performance seems to be degraded after upgrading to Kudu 0.10.0.
- The "remote bootstrap" process used to copy a tablet replica from one host to another has been renamed to "Tablet Copy". This resulted in the renaming of several RPC metrics. Any users previously explicitly fetching or monitoring metrics related to Remote Bootstrap should update their scripts to reflect the new names.
- The SparkSQL datasource for Kudu no longer supports mode `Overwrite`. Users should use the new `KuduContext.upsertRows` method instead. Additionally, inserts using the datasource are now upserts by default. The older behavior can be restored by setting the `operation` parameter to `insert`.

Kudu 0.9.1 Release Notes

Kudu 0.9.1 delivers incremental bug fixes over Kudu 0.9.0. It is fully compatible with Kudu 0.9.0. See also Issues resolved for Kudu 0.9.1 and Git changes between 0.9.0 and 0.9.1.

To upgrade Kudu to 0.9.1, see Upgrade Parcels or Upgrade Packages.

Issues Fixed in Kudu 0.9.1

- KUDU-1469 fixes a bug in Kudu's Raft consensus implementation that could cause a tablet to stop making progress after a leader election.
- Gerrit #3456 fixes a bug in which servers under high load could store metric information in incorrect memory locations, causing crashes or data corruption.
- Gerrit #3457 fixes a bug in which errors from the Java client would carry an incorrect error message.
- Other small bug fixes were backported to improve stability.

Kudu 0.9.0 Release Notes

Kudu 0.9.0 delivers incremental features, improvements, and bug fixes. See also Issues resolved for Kudu 0.9 and Git changes between 0.8.0 and 0.9.0.

To upgrade Kudu to 0.9.0, see Upgrade Parcels or Upgrade Packages.

New Features in Kudu 0.9.0

- KUDU-1306: Scan token API for creating partition-aware scan descriptors. This API simplifies executing parallel scans for clients and query engines.
- KUDU-1002: Added support for UPSERT operations, whereby a row is inserted if it does not yet exist, but updated if it does. Support for UPSERT is included in the Java, C++, and Python APIs, but not Impala.
- Gerrit 2848: Added a Kudu datasource for Spark. This datasource uses the Kudu client directly instead of using the MapReduce API. Predicate pushdowns for `spark-sql` and Spark filters are included, as well as parallel retrieval for multiple tablets and column projections. See an example of Kudu integration with Spark.
- Gerrit 2992: Added the ability to update and insert from Spark using a Kudu datasource.

Other Improvements and Changes in Kudu 0.9.0

All Kudu clients have longer default timeout values, as listed below.

Java

- The default operation timeout and the default admin operation timeout are now set to 30 seconds instead of 10.
- The default socket read timeout is now 10 seconds instead of 5.

C++

- The default admin timeout is now 30 seconds instead of 10.
- The default RPC timeout is now 10 seconds instead of 5.
- The default scan timeout is now 30 seconds instead of 15.

Some default settings related to I/O behavior during flushes and compactions have been changed:

- The default for `flush_threshold_mb` has been increased from 64 MB to 1000 MB.
- The default for `cfile_do_on_finish` has been changed from `close` to `flush`. Experiments using YCSB indicate that these values provide better throughput for write-heavy applications on typical server hardware.

- KUDU-1415: Added statistics in the Java client, such as the number of bytes written and the number of operations applied.
- KUDU-1451: Tablet servers take less time to restart when the tablet server must clean up many previously deleted tablets. Tablets are now cleaned up after they are deleted.

Issues Fixed in Kudu 0.9.0

- KUDU-678: Fixed a leak that occurred during `DiskRowSet` compactions where tiny blocks were still written to disk even if there were no REDO records. With the default block manager, this often resulted in block containers with thousands of tiny blocks.
- KUDU-1437: Fixed a data corruption issue that occurred after compacting sequences of negative INT32 values in a column that was configured with RLE encoding.

Incompatible Changes in Kudu 0.9.0

- The `KuduTableInputFormat` command has changed the way in which it handles scan predicates, including how it serializes predicates to the job configuration object. The new configuration key is `kudu.mapreduce.encoded.predicate`. Clients using the `TableInputFormatConfigurator` are not affected.
- The `kudu-spark` sub-project has been renamed to follow naming conventions for Scala. The new name is `kudu-spark_2.10`.
- Default table partitioning has been removed. **All tables must now be created with explicit partitioning.** Existing tables are unaffected. See the schema design guide for more details.

Limitations of Kudu 0.9.0

Kudu 0.9.0 has the same limitations as Kudu 0.8, listed in Limitations of Kudu 0.7.0 on page 26.

Upgrade Notes for Kudu 0.9.0

Before upgrading to Kudu 0.9.0, see Incompatible Changes in Kudu 0.9.0 on page 23.

Kudu 0.8.0 Release Notes

Kudu 0.8.0 delivers incremental features, improvements, and bug fixes over the previous versions. See also Issues resolved for Kudu 0.8 and Git changes between 0.7.1 and 0.8.0

To upgrade Kudu to 0.8.0, see Upgrade Parcels or Upgrade Packages

New Features in Kudu 0.8.0

- KUDU-431: A simple Flume sink has been implemented.

Other Improvements in Kudu 0.8.0

- KUDU-839: Java `RowError` now uses an `enum` error code.
- Gerrit 2138: The handling of column predicates has been re-implemented in the server and clients.
- KUDU-1379: Partition pruning has been implemented for C++ clients (but not yet for the Java client). This feature allows you to avoid reading a tablet if you know it does not serve the row keys you are querying.
- Gerrit 2641: Kudu now uses `earliest-deadline-first` RPC scheduling and rejection. This changes the behavior of the RPC service queue to prevent unfairness when processing a backlog of RPC threads and to increase the likelihood that an RPC will be processed before it can time out.
- Gerrit 2239: The concept of "feature flags" was introduced in order to manage compatibility between different Kudu versions. One case where this is helpful is if a newer client attempts to use a feature unsupported by the currently-running tablet server. Rather than receiving a cryptic error, the user gets an error message that is easier to interpret. This is an internal change for Kudu system developers and requires no action by users of the clients or API.

Issues Fixed in Kudu 0.8.0

- KUDU-1337: Tablets from tables that were deleted might be unnecessarily re-bootstrapped when the leader gets the notification to delete itself after the replicas do.
- KUDU-969: If a tablet server shuts down while compacting a rowset and receiving updates for it, it might immediately crash upon restart while bootstrapping that rowset's tablet.
- KUDU-1354: Due to a bug in the Kudu implementation of MVCC where row locks were released before the MVCC commit happened, flushed data would include out-of-order transactions, triggering a crash on the next compaction.
- KUDU-1322: The C++ client now retries write operations if the tablet it is trying to reach has already been deleted.
- Gerrit 2571: Due to a bug in the Java client, users were unable to close the `kudu-spark` shell because of lingering non-daemon threads.

Incompatible Changes in Kudu 0.8.0

0.8.0 clients are not fully compatible with servers running Kudu 0.7.1 or lower. In particular, scans that specify column predicates will fail. To work around this issue, upgrade all Kudu servers before upgrading clients.

Limitations of Kudu 0.8.0

Kudu 0.8.0 has the same limitations as Kudu 0.7.0, listed in Limitations of Kudu 0.7.0 on page 26.

Upgrade Notes for Kudu 0.8.0

Before upgrading to Kudu 0.8.0, see Incompatible Changes in Kudu 0.8.0 on page 24.

Kudu 0.7.1 Release Notes

Kudu 0.7.1 is a bug-fix release for 0.7.0. Users of Kudu 0.7.0 should upgrade to this version. See also Issues resolved for Kudu 0.7.1 and Git changes between 0.7.0 and 0.7.1.

To upgrade Kudu to 0.7.1, see Upgrade Parcels or Upgrade Packages.

Issues Fixed in Kudu 0.7.1

For a list of issues fixed in Kudu 0.7.1, see this JIRA query. The following notable fixes are included:

- KUDU-1325 fixes a tablet server crash that could occur during table deletion. In some cases, while a table was being deleted, other replicas would attempt to re-replicate tablets to servers that had already processed the deletion. This could trigger a race condition that caused a crash.
- KUDU-1341 fixes a potential data corruption and crash that could happen shortly after tablet server restarts in workloads that repeatedly delete and re-insert rows with the same primary key. In most cases, this corruption affected only a single replica and could be repaired by re-replicating from another.
- KUDU-1343 fixes a bug in the Java client that occurs when a scanner has to scan multiple batches from one tablet and then start scanning from another. In particular, this affected any scans using the Java client that read large numbers of rows from multi-tablet tables.
- KUDU-1345 fixes a bug where the hybrid clock could jump backwards, resulting in a crash followed by an inability to restart the affected tablet server.
- KUDU-1360 fixes a bug in the `kudu-spark` module that prevented reading rows with `NULL` values.

Limitations of Kudu 0.7.1

Kudu 0.7.1 has the same limitations as Kudu 0.7.0, listed in Limitations of Kudu 0.7.0 on page 26.

Upgrade Notes For Kudu 0.7.1

Kudu 0.7.1 has the same upgrade notes as Kudu 0.7.0, listed in Upgrade Notes For Kudu 0.7.0 on page 28.

Kudu 0.7.0 Release Notes

Kudu 0.7.0 is the first release as part of the Apache Incubator and includes a number of changes, new features, improvements, and fixes. See also Issues resolved for Kudu 0.7.0 and Git changes between 0.6.0 and 0.7.0.

To upgrade Kudu to 0.7, see Upgrade Parcels or Upgrade Packages.

New Features in Kudu 0.7.0

Initial work for Spark integration

With the goal of Spark integration, a new `kuduRDD` API has been added that wraps `newAPIHadoopRDD` and includes a default source for Spark SQL.

Other Improvements in Kudu 0.7.0

- Support for RHEL 7, CentOS 7, and SLES 12 has been added.
- The Python client is no longer considered experimental.
- The file block manager performance is improved, but it is still not recommended for real-world use.
- The master now attempts to spread tablets more evenly across the cluster during table creation. This has no impact on existing tables, but improves the speed at which under-replicated tablets are re-replicated after a tablet server failure.
- All licensing documents have been modified to adhere to ASF guidelines.

- The C++ client library is now explicitly built against the old GCC 5 ABI. If you use gcc5 to build a Kudu application, your application must use the old ABI as well. This is typically achieved by defining the `_GLIBCXX_USE_CXX11_ABI` macro at compile time when building your application. For more information, see GCC 5 and the C++ 11 ABI.

Issues Fixed in Kudu 0.7.0

For a list of issues fixed in Kudu 0.7, see this JIRA query.

Incompatible Changes in Kudu 0.7.0

- The C++ client includes a new API, `KuduScanBatch`, which performs better when a large number of small rows are returned in a batch. The old API of `vector<KuduRowResult>` is deprecated.

> **Note:** This change is API-compatible but **not** ABI-compatible.

- The default replication factor has been changed from 1 to 3. Existing tables continue to use the replication factor they were created with. Applications that create tables may not work properly if they assume a replication factor of 1 and fewer than 3 replicas are available. To use the previous default replication factor, start the master with the configuration flag `--default_num_replicas=1`.
- The Python client has been rewritten, with a focus on improving code quality and testing. The read path (scanners) has been improved by adding many of the features already supported by the C++ and Java clients. The Python client is no longer considered experimental.

Limitations of Kudu 0.7.0

Operating System Limitations

- RHEL 7 or 6.4 or newer, CentOS 7 or 6.4 or newer, and Ubuntu Trusty are the only operating systems supported for installation in the public beta. Others may work but have not been tested. You can build Kudu from source on SLES 12, but binaries are not provided.

Storage Limitations

- Kudu has been tested with up to 4 TB of data per tablet server. More testing is needed for denser storage configurations.

Schema Limitations

- Testing with more than 20 columns has been limited.
- Multi-kilobyte rows have not been thoroughly tested.
- The columns that make up the primary key must be listed first in the schema.
- Key columns cannot be altered. You must drop and re-create a table to change its keys.
- Key columns must not be null.
- Columns with `DOUBLE`, `FLOAT`, or `BOOL` types are not allowed as part of a primary key definition.
- Type and nullability of existing columns cannot be changed by altering the table.
- A table's primary key cannot be changed.
- Dropping a column does not immediately reclaim space.; compaction must run first. You cannot run compaction manually. Dropping the table reclaims space immediately.

Ingest Limitations

- Ingest through Sqoop or Flume is not supported in the public beta. For bulk ingest, use Impala's `CREATE TABLE AS SELECT` functionality or use Kudu's Java or C++ API.
- Tables must be manually pre-split into tablets using simple or compound primary keys. Automatic splitting is not yet possible. Instead, add split rows at table creation.

- Tablets cannot currently be merged. Instead, create a new table with the contents of the old tables to be merged.

Cloudera Manager Limitations

- Some metrics, such as latency histograms, are not yet available in Cloudera Manager.
- Some service and role chart pages are still under development. More charts and metrics will be visible in future releases.

Replication and Backup Limitations

- Replication and failover of Kudu masters is considered experimental. Cloudera recommends running a single master and periodically perform a manual backup of its data directories.

Impala Limitations

- To use Kudu with Impala, you must install a special release of Impala. Obtaining and installing a compatible Impala release is detailed in Using Apache Impala (incubating) with Kudu on page 56.
- To use Impala_Kudu alongside an existing Impala instance, you must install using parcels.
- Updates, inserts, and deletes through Impala are nontransactional. If a query fails, any partial effects are not be rolled back.
- All queries are distributed across all Impala nodes that host a replica of the target table(s), even if a predicate on a primary key could correctly restrict the query to a single tablet. This limits the maximum concurrency of short queries made through Impala.
- Timestamp and decimal type are not supported.
- The maximum parallelism of a single query is limited to the number of tablets in a table. To optimize analytic performance, spread your data across 10 or more tablets per host for a large table.
- Impala can push down only predicates involving =, <=, >=, or `BETWEEN` comparisons between a column and a literal value. Impala pushes down predicates < and > for integer columns only. For example, for a table with an integer key `ts`, and a string key `name`, the predicate `WHERE ts >= 12345` converts to an efficient range scan, whereas `WHERE name > smith` currently fetches all data from the table and evaluates the predicate within Impala.

Security Limitations

- Authentication and authorization are not included in the public beta.
- Data encryption is not included in the public beta.

Client and API Limitations

- Potentially incompatible C++ and Java API changes may be required during the public beta.
- `ALTER TABLE` is not yet fully supported through the client APIs. More `ALTER TABLE` operations will be available in future betas.

Application Integration Limitations

- The Spark DataFrame implementation is not yet complete.

Other Known Issues

The following are known bugs and issues with the current beta release. They will be addressed in later beta releases.

- Building Kudu from source using `gcc` 4.6 or 4.7 causes runtime and test failures. Be sure you are using a different version of `gcc` if you build Kudu from source.
- If the Kudu master is configured with the `-log_fsync_all` option, tablet servers and clients will experience frequent timeouts, and the cluster may become unusable.

- If a tablet server has a very large number of tablets, it may take several minutes to start up. Limit the number of tablets per server to 100 or fewer, and consider this limitation when pre-splitting your tables. If you notice slow start-up times, you can monitor the number of tablets per server in the web UI.

Upgrade Notes For Kudu 0.7.0

- Kudu 0.7.0 maintains wire compatibility with Kudu 0.6.0. A Kudu 0.7.0 client can communicate with a Kudu 0.6.0 cluster, and vice versa. For that reason, you do not need to upgrade client JARs at the same time the cluster is upgraded.
- The same wire compatibility guarantees apply to the `Impala_Kudu` fork that was released with Kudu 0.5.0.
- Review Incompatible Changes in Kudu 0.7.0 on page 26 before upgrading to Kudu 0.7.

See Upgrading Kudu for instructions.

Kudu 0.6 Release Notes

To upgrade Kudu to 0.6, see Upgrade Parcels or Upgrade Packages.

New Features in Kudu 0.6

Row Error Reporting

The Java client includes new methods `countPendingErrors()` and `getPendingErrors()` on `KuduSession`. These methods allow you to count and retrieve outstanding row errors when configuring sessions with `AUTO_FLUSH_BACKGROUND`.

New Server-Side Metrics

New server-level metrics allow you to monitor CPU usage and context switching.

Issues Fixed in Kudu 0.6

For a list of issues addressed in Kudu 0.6, see this JIRA query.

Limitations of Kudu 0.6

Operating System Limitations

- RHEL 6.4 or newer, CentOS 6.4 or newer, and Ubuntu Trusty are the only operating systems supported for installation in the public beta. Others may work but have not been tested.

Storage Limitations

- Kudu has been tested with up to 4 TB of data per tablet server. More testing is needed for denser storage configurations.

Schema Limitations

- Testing with more than 20 columns has been limited.
- Multi-kilobyte rows have not been thoroughly tested.
- The columns which make up the primary key must be listed first in the schema.
- Key columns cannot be altered. You must drop and recreate a table to change its keys.
- Key columns must not be null.
- Columns with `DOUBLE`, `FLOAT`, or `BOOL` types are not allowed as part of a primary key definition.
- Type and nullability of existing columns cannot be changed by altering the table.
- A table’s primary key cannot be changed.
- Dropping a column does not immediately reclaim space. Compaction must run first. There is no way to run compaction manually, but dropping the table will reclaim the space immediately.

Ingest Limitations

- Ingest using Sqoop or Flume is not supported in the public beta. The recommended approach for bulk ingest is to use Impala's `CREATE TABLE AS SELECT` functionality or use the Kudu's Java or C++ API.
- Tables must be manually pre-split into tablets using simple or compound primary keys. Automatic splitting is not yet possible. Instead, add split rows at table creation.
- Tablets cannot currently be merged. Instead, create a new table with the contents of the old tables to be merged.

Cloudera Manager Limitations

- Some metrics, such as latency histograms, are not yet available in Cloudera Manager.
- Some service and role chart pages are still under development. More charts and metrics will be visible in future releases.

Replication and Backup Limitatinos

- Replication and failover of Kudu masters is considered experimental. It is recommended to run a single master and periodically perform a manual backup of its data directories.

Impala Limitations

- To use Kudu with Impala, you must install a special release of Impala. Obtaining and installing a compatible Impala release is detailed in Using Apache Impala (incubating) with Kudu on page 56.
- To use Impala_Kudu alongside an existing Impala instance, you must install using parcels.
- Updates, inserts, and deletes using Impala are non-transactional. If a query fails part of the way through, its partial effects will not be rolled back.
- All queries will be distributed across all Impala nodes which host a replica of the target table(s), even if a predicate on a primary key could correctly restrict the query to a single tablet. This limits the maximum concurrency of short queries made using Impala.
- No timestamp and decimal type support.
- The maximum parallelism of a single query is limited to the number of tablets in a table. For good analytic performance, aim for 10 or more tablets per host or large tables.
- Impala is only able to push down predicates involving =, <=, >=, or `BETWEEN` comparisons between a column and a literal value. Impala pushes down predicates < and > for integer columns only. For example, for a table with an integer key `ts`, and a string key `name`, the predicate `WHERE ts >= 12345` will convert into an efficient range scan, whereas `WHERE name > smith` will currently fetch all data from the table and evaluate the predicate within Impala.

Security Limitations

- Authentication and authorization are not included in the public beta.
- Data encryption is not included in the public beta.

Client and API Limitations

- Potentially-incompatible C++ and Java API changes may be required during the public beta.
- `ALTER TABLE` is not yet fully supported using the client APIs. More `ALTER TABLE` operations will become available in future betas.
- The Python API is not supported.

Application Integration Limitations

- The Spark DataFrame implementation is not yet complete.

Other Known Issues

The following are known bugs and issues with the current beta release. They will be addressed in later beta releases.

- Building Kudu from source using `gcc` 4.6 or 4.7 causes runtime and test failures. Be sure you are using a different version of `gcc` if you build Kudu from source.
- If the Kudu master is configured with the `-log_fsync_all` option, tablet servers and clients will experience frequent timeouts, and the cluster may become unusable.
- If a tablet server has a very large number of tablets, it may take several minutes to start up. It is recommended to limit the number of tablets per server to 100 or fewer. Consider this limitation when pre-splitting your tables. If you notice slow start-up times, you can monitor the number of tablets per server in the web UI.

Upgrade Notes For Kudu 0.6

- Kudu 0.6.0 maintains wire compatibility with Kudu 0.5.0. This means that a Kudu 0.6.0 client can communicate with a Kudu 0.5.0 cluster, and vice versa. For that reason, you do not need to upgrade client JARs at the same time the cluster is upgraded.
- The same wire compatibility guarantees apply to the Impala_Kudu fork that was released with Kudu 0.5.0 and 0.6.0.
- The Kudu 0.6.0 client API is not compatible with the Kudu 0.5.0 client API. See the Kudu 0.6.0 release notes for details.

See Upgrading Kudu for instructions.

Kudu 0.5 Release Notes

Limitations of Kudu 0.5

Operating System Limitations

- RHEL 6.4 or newer, CentOS 6.4 or newer, and Ubuntu Trusty are the only operating systems supported for installation in the public beta. Others may work but have not been tested.

Storage Limitations

- Kudu has been tested with up to 4 TB of data per tablet server. More testing is needed for denser storage configurations.

Schema Limitations

- Testing with more than 20 columns has been limited.
- Multi-kilobyte rows have not been thoroughly tested.
- The columns which make up the primary key must be listed first in the schema.
- Key columns cannot be altered. You must drop and recreate a table to change its keys.
- Key columns must not be null.
- Columns with `DOUBLE`, `FLOAT`, or `BOOL` types are not allowed as part of a primary key definition.
- Type and nullability of existing columns cannot be changed by altering the table.
- A table’s primary key cannot be changed.
- Dropping a column does not immediately reclaim space. Compaction must run first. There is no way to run compaction manually, but dropping the table will reclaim the space immediately.

Ingest Limitations

- Ingest using Sqoop or Flume is not supported in the public beta. The recommended approach for bulk ingest is to use Impala’s `CREATE TABLE AS SELECT` functionality or use the Kudu's Java or C++ API.
- Tables must be manually pre-split into tablets using simple or compound primary keys. Automatic splitting is not yet possible. Instead, add split rows at table creation.
- Tablets cannot currently be merged. Instead, create a new table with the contents of the old tables to be merged.

Cloudera Manager Limitations

- Some metrics, such as latency histograms, are not yet available in Cloudera Manager.
- Some service and role chart pages are still under development. More charts and metrics will be visible in future releases.

Replication and Backup Limitatinos

- Replication and failover of Kudu masters is considered experimental. It is recommended to run a single master and periodically perform a manual backup of its data directories.

Impala Limitations

- To use Kudu with Impala, you must install a special release of Impala. Obtaining and installing a compatible Impala release is detailed in Using Apache Impala (incubating) with Kudu on page 56.
- To use Impala_Kudu alongside an existing Impala instance, you must install using parcels.
- Updates, inserts, and deletes using Impala are non-transactional. If a query fails part of the way through, its partial effects will not be rolled back.
- All queries will be distributed across all Impala nodes which host a replica of the target table(s), even if a predicate on a primary key could correctly restrict the query to a single tablet. This limits the maximum concurrency of short queries made using Impala.
- No timestamp and decimal type support.
- The maximum parallelism of a single query is limited to the number of tablets in a table. For good analytic performance, aim for 10 or more tablets per host or large tables.
- Impala is only able to push down predicates involving =, <=, >=, or `BETWEEN` comparisons between a column and a literal value. Impala pushes down predicates < and > for integer columns only. For example, for a table with an integer key `ts`, and a string key `name`, the predicate `WHERE ts >= 12345` will convert into an efficient range scan, whereas `WHERE name > smith` will currently fetch all data from the table and evaluate the predicate within Impala.

Security Limitations

- Authentication and authorization are not included in the public beta.
- Data encryption is not included in the public beta.

Client and API Limitations

- Potentially-incompatible C++ and Java API changes may be required during the public beta.
- `ALTER TABLE` is not yet fully supported using the client APIs. More `ALTER TABLE` operations will become available in future betas.
- The Python API is not supported.

Application Integration Limitations

- The Spark DataFrame implementation is not yet complete.

Other Known Issues

The following are known bugs and issues with the current beta release. They will be addressed in later beta releases.

- Building Kudu from source using `gcc` 4.6 or 4.7 causes runtime and test failures. Be sure you are using a different version of `gcc` if you build Kudu from source.
- If the Kudu master is configured with the `-log_fsync_all` option, tablet servers and clients will experience frequent timeouts, and the cluster may become unusable.
- If a tablet server has a very large number of tablets, it may take several minutes to start up. It is recommended to limit the number of tablets per server to 100 or fewer. Consider this limitation when pre-splitting your tables. If you notice slow start-up times, you can monitor the number of tablets per server in the web UI.

Next Steps

Kudu Quickstart

Installing Kudu

Configuring Kudu

Apache Kudu Schema Design and Usage Limitations

The following sections describe known issues and limitations in Kudu, as of the current release.

Schema Design Limitations

Primary Key

The columns which make up the primary key must be listed first in the schema.

- The columns which make up the primary key must be listed first in the schema.
- Columns with `DOUBLE`, `FLOAT`, or `BOOL` types are not allowed as part of a primary key definition. Additionally, all columns that are part of a primary key definition must be `NOT NULL`.
- The primary key of a row may not be modified using the `UPDATE` functionality. To modify a row's primary key, the row must be deleted and re-inserted with the modified key. Such a modification is non-atomic.
- Columns that are part of the primary key cannot be renamed. The primary key may not be changed after the table is created. You must drop and recreate a table to select a new primary key or rename key columns.

Valid Identifiers

Identifiers such as table and column names must be valid UTF-8 sequences and no longer than 256 bytes.

Number of Columns

By default, Kudu will not permit the creation of tables with more than 300 columns. We recommend schema designs that use fewer columns for best performance.

Size of Cells

No individual cell may be larger than 64KB. The cells making up a composite key are limited to a total of 16KB after the internal composite-key encoding done by Kudu. Inserting rows not conforming to these limitations will result in errors being returned to the client.

Size of Rows

Kudu was primarily designed for analytic use cases. Although individual cells may be up to 64KB, and Kudu supports up to 300 columns, it is recommended that no single row be larger than a few hundred KB. You are likely to encounter issues if a single row contains multiple kilobytes of data.

Non-alterable Partitioning

Kudu does not allow you to change how a table is partitioned after creation, with the exception of adding or dropping range partitions.

Non-alterable Column Types

Kudu does not allow the type or nullability of an existing column to be altered.

Partition Splitting

Partitions cannot be split or merged after table creation.

Dropping Columns and Tables

Dropping a column does not immediately reclaim space. Compaction must run first. There is no way to run compaction manually, but dropping the table will reclaim the space immediately.

If you are using Apache Impala (incubating) to query Kudu tables, refer the section on Impala Integration Limitations on page 33 as well.

Partitioning Limitations

- Tables must be manually pre-split into tablets using simple or compound primary keys. Automatic splitting is not yet possible. Range partitions may be added or dropped after a table has been created. See Apache Kudu Schema Design on page 66 for more information.
- Data in existing tables cannot currently be automatically repartitioned. As a workaround, create a new table with the new partitioning and insert the contents of the old table.

Replication and Backup Limitations

- Kudu does not currently include any built-in features for backup and restore. Users are encouraged to use tools such as Spark or Impala to export or import tables as necessary.

Impala Integration Limitations

- When creating a Kudu table, the `CREATE TABLE` statement must include the primary key columns before other columns, in primary key order.
- Impala cannot update values in primary key columns.
- Impala cannot create Kudu tables with `TIMESTAMP`, `DECIMAL`, `VARCHAR`, or nested-typed columns.
- Kudu tables with a name containing upper case or non-ASCII characters must be assigned an alternate name when used as an external table in Impala.
- Kudu tables with a column name containing upper case or non-ASCII characters may not be used as an external table in Impala. Non-primary key columns may be renamed in Kudu to work around this issue.
- Kudu tables containing `UNIXTIME_MICROS`-typed columns may not be used as an external table in Impala.
- `NULL`, `NOT NULL`, `!=`, and `LIKE` predicates are not pushed to Kudu, and instead will be evaluated by the Impala scan node. This may decrease performance relative to other types of predicates.
- Updates, inserts, and deletes using Impala are non-transactional. If a query fails part of the way through, its partial effects will not be rolled back.
- The maximum parallelism of a single query is limited to the number of tablets in a table. For good analytic performance, aim for 10 or more tablets per host or use large tables.

Impala Keywords Not Supported for Creating Kudu Tables

- `PARTITIONED`
- `LOCATION`
- `ROWFORMAT`

Spark Integration Limitations

- Kudu tables with a name containing upper case or non-ASCII characters must be assigned an alternate name when registered as a temporary table.
- Kudu tables with a column name containing upper case or non-ASCII characters may not be used with SparkSQL. Non-primary key columns may be renamed in Kudu to work around this issue.

- `NULL`, `NOT NULL`, <>, `OR`, `LIKE`, and `IN` predicates are not pushed to Kudu, and instead will be evaluated by the Spark task.
- Kudu does not support all types supported by Spark SQL, such as `Date`, `Decimal` and complex types.

Security Limitations

- Authentication and authorization features are not implemented.
- Data encryption is not built in. Kudu has been reported to run correctly on systems using local block device encryption (e.g. `dmcrypt`).

Other Known Issues

The following are known bugs and issues with the current release of Kudu. They will be addressed in later releases. Note that this list is not exhaustive, and is meant to communicate only the most important known issues.

- If the Kudu master is configured with the `-log_force_fsync_all` option, tablet servers and clients will experience frequent timeouts, and the cluster may become unusable.
- If a tablet server has a very large number of tablets, it may take several minutes to start up. It is recommended to limit the number of tablets per server to 100 or fewer. Consider this limitation when pre-splitting your tables. If you notice slow start-up times, you can monitor the number of tablets per server in the web UI.

Installing and Upgrading Apache Kudu

You can install Apache Kudu in a cluster managed by Cloudera Manager, using either parcels or packages. If you do not use Cloudera Manager, you can install Kudu using packages.

Tip: To start using Kudu in minutes, without the need to install anything, see the Kudu Quickstart documentation.

Kudu Installation Requirements

- **Hardware**
  - One or more hosts to run Kudu masters. You should have either one master (provides no fault tolerance), three masters (can tolerate one failure), or five masters (can tolerate two failures).
  - One or more hosts to run Kudu tablet servers. With replication, a minimum of three tablet servers is necessary.
- **Operating systems**
  - **Linux**
    - RHEL/CentOS 6.4, 6.5, 6.6, 6.7, 6.8, 7.1, 7.2, 7.3
    - Oracle Linux (OL) 6.4, 6.5, 6.6, 6.7, 6.8, 7.1, 7.2, 7.3
    - Ubuntu 14.04 (Trusty), 16.04 (Xenial)
    - Debian 8.2, 8.4 (Jessie)
    - SLES 12 Service Pack 1
    - A kernel version and filesystem that support hole punching. Hole punching is the use of the `fallocate(2)` system call with the `FALLOC_FL_PUNCH_HOLE` option set. See Error during hole punch test on page 77. If you cannot meet this requirement, see this workaround.
    - NTP
  - **MacOS**
    - OS X 10.10 Yosemite, OS X 10.11 El Capitan, and macOS Sierra.
    - Pre-built macOS packages are not provided.
  - **Windows**
    - Microsoft Windows is not supported.
- **Management -** To manage Kudu with Cloudera Manager, Cloudera Manager 5.10.0 or later and CDH 5.10.0 or later are required.

  Note: Kudu is not supported in Cloudera Manager's single-user mode.

- **Storage -** If solid state storage is available, storing Kudu WALs on such high-performance media may significantly improve latency when Kudu is configured for its highest durability levels.

Install Kudu Using Cloudera Manager

To install and manage Kudu using Cloudera Manager, first download the Custom Service Descriptor (CSD) file for Kudu and upload it to `/opt/cloudera/csd/` on the Cloudera Manager server. Restart the Cloudera Manager server using the following operating system command.

```
$ sudo service cloudera-scm-server restart
```

Next, follow the instructions in either Install Kudu Using Parcels on page 36 or Install Kudu Using Packages on page 37.

Install Kudu Using Parcels

After uploading the CSD file for Kudu and restarting the Cloudera Manager server, follow these steps to install Kudu using parcels.

1. In Cloudera Manager, go to **Hosts** > **Parcels**. Find KUDU in the list, and click **Download**.
2. When the download is complete, select your cluster from the **Locations** selector, and click **Distribute**. If you only have one cluster, it is selected automatically.
3. When distribution is complete, click **Activate** to activate the parcel. Restart the cluster when prompted. This may take several minutes.
4. Install the Kudu service on your cluster. Go to the cluster where you want to install Kudu. Click **Actions** > **Add a Service**. Select **Kudu** from the list, and click **Continue**.
5. Select a host for the master role and one or more hosts for the tablet server roles. A host can act as both a master and a tablet server, but this may cause performance problems on a large cluster. The Kudu master process is not resource-intensive and can be collocated with other similar processes such as the HDFS Namenode or YARN ResourceManager. After selecting hosts, click **Continue**.
6. Configure the storage locations for Kudu data and write-ahead log (WAL) files on masters and tablet servers. Cloudera Manager will create the directories.
    - You can use the same directory to store data and WALs.
    - You cannot store WALs in a subdirectory of the data directory.
    - If any host is both a master and tablet server, configure different directories for master and tablet server. For instance, `/data/kudu/master` and `/data/kudu/tserver`.
    - If you choose a filesystem that does not support hole punching, service start-up will fail. **Only** if service start-up fails for this reason, exit the wizard by clicking the Cloudera logo at the top left, and enable the file block manager. This is not appropriate for production. See Enabling the File Block Manager on page 36.
7. If your filesystem does support hole punching, do not exit the wizard. Click **Continue**. Kudu masters and tablet servers are started. Otherwise, go to the **Kudu** service, and click **Actions** > **Start**.
8. Verify that services are running using one of the following methods:
    - Examine the output of the `ps` command on servers to verify one or both of `kudu-master` or `kudu-tserver` processes is running.
    - Access the master or tablet server web UI by opening the URL in your web browser. The URL is `http://<_host_name_>:8051/` for masters or `http://<_host_name_>:8050/` for tablet servers.
9. Restart the Service Monitor to begin generating health checks and charts for Kudu. Go to the **Cloudera Manager** service and click **Service Monitor.** Choose **Actions** > **Restart**.
10. To manage roles, go to the **Kudu** service and use the **Actions** menu to stop, start, restart, or otherwise manage the service.

Enabling the File Block Manager

If your underlying filesystem does not support hole punching, Kudu will not start unless you enable the file block manager. This is not appropriate for production systems. If your filesystem does support hole punching, there is no reason to use the file block manager.

> **Note:** The file block manager does not perform well at scale and should only be used for small-scale development and testing.

1. If you are still in the Cloudera configuration wizard, exit the configuration wizard by clicking the Cloudera logo at the top of the Cloudera Manager interface.
2. Go to the **Kudu** service.

3. Go to **Configuration** and search for the **Kudu Service Advanced Configuration Snippet (Safety Valve) for gflagfile** configuration option.
4. Add the following line to it, and save your changes:

```
--block_manager=file
```

Install Kudu Using Packages

If you use packages with Cloudera Manager, follow these instructions after uploading the CSD file for Kudu and restarting the Cloudera Manager server.

Table 1: Kudu Repository and Package Links

Operating System	Repository Package	Individual Packages
RHEL	RHEL 6 or RHEL 7	RHEL 6
Ubuntu	Trusty, Xenial	Trusty, Xenial
SLES	SLES 12	SLES 12
Debian	Jessie	Jessie

1. Cloudera recommends installing the Kudu repositories for your operating system. Use the links in Table 1: Kudu Repository and Package Links on page 37 to download the appropriate repository installer. Save the repository installer to `/etc/yum.repos.d/` for RHEL, `/etc/apt/sources.list.d/` for Ubuntu/Debian, or `/etc/zypp/repos.d` for SLES.

   - If you use Cloudera Manager, you only need to install the kudu package:

```
$ sudo yum install kudu
```

```
$ sudo apt-get install kudu
```

   - If you need the C++ client development libraries or the Kudu SDK, install `kudu-client` and `kudu-client-devel` packages for RHEL, or `libkuduclient0` and `libkuduclient-dev` packages for Ubuntu.
   - If you use Cloudera Manager, **do not** install the `kudu-master` and `kudu-tserver` packages, which provide operating system startup scripts for using Kudu without Cloudera Manager.

2. Install the Kudu service on your cluster. Go to the cluster where you want to install Kudu. Click **Actions** > **Add a Service**. Select **Kudu** from the list, and click **Continue**.
3. Select a host for the master role and one or more hosts for the tablet server roles. A host can act as both a master and a tablet server, but this may cause performance problems on a large cluster. The Kudu master process is not resource-intensive and can be collocated with other similar processes such as the HDFS Namenode or YARN ResourceManager. After selecting hosts, click **Continue**.
4. Configure the storage locations for Kudu data and write-ahead log (WAL) files on masters and tablet servers. Cloudera Manager will create the directories.

   - You can use the same directory to store data and WALs.
   - You cannot store WALs in a subdirectory of the data directory.
   - If any host is both a master and tablet server, configure different directories for master and tablet server. For instance, `/data/kudu/master` and `/data/kudu/tserver`.
   - If you choose a filesystem that does not support hole punching, service start-up will fail. **Only** if service start-up fails for this reason, exit the wizard by clicking the Cloudera logo at the top left, and enable the file block manager. This is not appropriate for production. See Enabling the File Block Manager on page 36.

5. If your filesystem does support hole punching, do not exit the wizard. Click **Continue**. Kudu masters and tablet servers are started. Otherwise, go to the **Kudu** service, and click **Actions** > **Start**.
6. Verify that services are running using one of the following methods:
   - Examine the output of the `ps` command on servers to verify one or both of `kudu-master` or `kudu-tserver` processes is running.
   - Access the master or tablet server web UI by opening the URL in your web browser. The URL is `http://<_host_name_>:8051/` for masters or `http://<_host_name_>:8050/` for tablet servers.
7. To manage roles, go to the **Kudu** service and use the **Actions** menu to stop, start, restart, or otherwise manage the service.
8. Restart the Service Monitor to begin generating health checks and charts for Kudu. Go to the **Cloudera Manager** service and click **Service Monitor.** Choose **Actions** > **Restart**.

Install Kudu Using the Command Line

> **Important:** If you use Cloudera Manager, do not use these command-line instructions.

Follow these steps on each node which will participate in your Kudu cluster.

1. Cloudera recommends installing the Kudu repositories for your operating system. Use the links in Table 1: Kudu Repository and Package Links on page 37 to download the appropriate repository installer.
   - Install the kudu package, using the appropriate commands for your operating system:

```
$ sudo yum install kudu
```

```
$ sudo apt-get install kudu
```

   - If you need the C++ client development libraries or the Kudu SDK, install `kudu-client` and `kudu-client-devel` packages for RHEL, or `libkuduclient0` and `libkuduclient-dev` packages for Ubuntu.
   - Install the `kudu-master` and `kudu-tserver` packages, which provide operating system start-up scripts for the Kudu master and tablet servers.
2. The packages create a `kudu-conf` entry in the operating system's alternatives database, and they ship the built-in `conf.dist` alternative. To adjust your configuration, you can either edit the files in `/etc/kudu/conf/` directly, or create a new alternative using the operating system utilities. If you create a new alternative, make sure the alternative is the directory pointed to by the `/etc/kudu/conf/` symbolic link, and create custom configuration files there. Some parts of the configuration are configured in `/etc/default/kudu-master` and `/etc/default/kudu-tserver` files as well. You should include or duplicate these configuration options if you create custom configuration files.

   Review the configuration, including the default WAL and data directory locations, and adjust them according to your requirements.
3. Start Kudu services using the following commands on the appropriate nodes:

```
$ sudo service kudu-master start
$ sudo service kudu-tserver start
```

4. To stop Kudu services, use the following commands:

```
$ sudo service kudu-master stop
$ sudo service kudu-tserver stop
```

5. Configure the Kudu services to start automatically when the server starts, by adding them to the default runlevel.

```
$ sudo chkconfig kudu-master on              # RHEL / CentOS
$ sudo chkconfig kudu-tserver on             # RHEL / CentOS

$ sudo update-rc.d kudu-master defaults      # Ubuntu
$ sudo update-rc.d kudu-tserver defaults     # Ubuntu
```

Upgrading Kudu

Before upgrading Kudu, make sure read the Release Notes relevant to the version you are upgrading to.

Upgrading Kudu Using Parcels

To upgrade Kudu, use the following instructions if you use Cloudera Manager. If you do not use Cloudera Manager, see the instructions for upgrading Kudu packages.

1. First, download the Custom Service Descriptor (CSD) file for Kudu and upload it to `/opt/cloudera/csd/` on the Cloudera Manager server. Restart the Cloudera Manager server using the following operating system command.

```
$ sudo service cloudera-scm-server restart
```

2. Go to **Hosts**. Click **Parcels**.
3. Click **Check For New Parcels**.
4. Find the new version of Kudu in the list of parcels. Download, distribute, and activate it on your cluster.

Upgrade Kudu Using Packages

To upgrade Kudu, use the following instructions if you use Cloudera Manager. If you do not use Cloudera Manager, see the instructions for upgrading Kudu packages.

Using RHEL:

1. First, download the Custom Service Descriptor (CSD) file for Kudu and upload it to `/opt/cloudera/csd/` on the Cloudera Manager server. Restart the Cloudera Manager server using the following operating system command.

```
$ sudo service cloudera-scm-server restart
```

2. Stop the Kudu service in Cloudera Manager. Go to the Kudu service and select **Actions** > **Stop**.
3. Issue the following commands at the command line on each Kudu host:

```
$ sudo yum -y clean all
$ sudo yum -y upgrade kudu
```

4. Start the Kudu service in Cloudera Manager. Go to the Kudu service and select **Actions** > **Start**.

Using Ubuntu:

1. First, download the Custom Service Descriptor (CSD) file for Kudu and upload it to `/opt/cloudera/csd/` on the Cloudera Manager server. Restart the Cloudera Manager server using the following operating system command.

```
$ sudo service cloudera-scm-server restart
```

2. If you use a repository, re-download the repository list file to ensure that you have the latest information. See Table 1: Kudu Repository and Package Links on page 37.
3. Stop the Kudu service in Cloudera Manager. Go to the Kudu service and select **Actions** > **Stop**.

4. Issue the following commands at the command line on each Kudu host:

```
$ sudo apt-get update
$ sudo apt-get install kudu
```

5. Start the Kudu service in Cloudera Manager. Go to the Kudu service and select **Actions** > **Start**.

Next Steps

Read about Using Apache Impala (incubating) with Kudu on page 56.

For more information about using Kudu, go to the Kudu project page, where you can find official documentation, links to the Github repository and examples, and other resources.

For a reading list and other helpful links, refer to More Resources for Apache Kudu on page 106.

Apache Kudu Configuration

To configure the behavior of each Kudu process, you can pass command-line flags when you start it, or read those options from configuration files by passing them using one or more `--flagfile=<file>` options. You can even include the `--flagfile` option within your configuration file to include other files. Learn more about gflags by reading its documentation.

You can place options for masters and tablet servers into the same configuration file, and each will ignore options that do not apply.

Flags can be prefixed with either one or two `-` characters. This documentation standardizes on two: `--example_flag`.

Only the most common configuration options are documented in this topic. For a more exhaustive list of configuration options, see the Kudu Configuration Reference. To see all configuration flags for a given executable, run it with the `--help` option.

Experimental Flags

Some configuration flags are marked 'unsafe' and 'experimental'. Such flags are disabled by default. You may access these flags by enabling the additional flags, `--unlock_unsafe_flags` and `--unlock_experimental_flags`. Note that these flags might be removed or modified without a deprecation period or any prior notice in future Kudu releases. Cloudera does not support using unsafe and experimental flags. As a rule of thumb, Cloudera will not support any configuration flags not explicitly documented in the Kudu Configuration Reference Guide.

Configuring the Kudu Master

To see all available configuration options for the `kudu-master` executable, run it with the `--help` option:

```
$ kudu-master --help
```

Table 2: Supported Configuration Flags for Kudu Masters

Flag	Valid Options	Default	Description
`--master_addresses`	string	`localhost`	Comma-separated list of all the RPC addresses for Master consensus-configuration. If not specified, assumes a standalone Master.
`--fs_data_dirs`	string		Comma-separated list of directories where the Master will place its data blocks.
`--fs_wal_dir`	string		The directory where the Master will place its write-ahead logs. May be the same as *one of* the directories listed in `--fs_data_dirs`, but not a sub-directory of a data directory.

Flag	Valid Options	Default	Description
`--log_dir`	string	`/tmp`	The directory to store Master log files.

For the complete list of flags for masters, see the Kudu Master Configuration Reference.

Configuring Tablet Servers

To see all available configuration options for the `kudu-tserver` executable, run it with the `--help` option:

```
$ kudu-tserver --help
```

Table 3: Supported Configuration Flags for Kudu Tablet Servers

Flag	Valid Options	Default	Description
--fs_data_dirs	string		Comma-separated list of directories where the Tablet Server will place its data blocks.
--fs_wal_dir	string		The directory where the Tablet Server will place its write-ahead logs. May be the same as *one of* the directories listed in `--fs_data_dirs`, but not a sub-directory of a data directory.
--log_dir	string	/tmp	The directory to store Tablet Server log files
--tserver_master_addrs	string	`127.0.0.1:7051`	Comma separated addresses of the masters which the tablet server should connect to. The masters do not read this flag.
--block_cache_capacity_mb	integer	512	Maximum amount of memory allocated to the Kudu Tablet Server's block cache.
--memory_limit_hard_bytes	integer	4294967296	Maximum amount of memory a Tablet Server can consume before it starts rejecting all incoming writes.

For the complete list of flags for tablet servers, see the Kudu Tablet Server Configuration Reference.

Apache Kudu Administration

This topic describes how to perform common administrative tasks and workflows with Apache Kudu.

Starting and Stopping Kudu Processes

Start Kudu services using the following commands:

```
$ sudo service kudu-master start
$ sudo service kudu-tserver start
```

To stop Kudu services, use the following commands:

```
$ sudo service kudu-master stop
$ sudo service kudu-tserver stop
```

Kudu Web Interfaces

Kudu tablet servers and masters expose useful operational information on a built-in web interface.

Kudu Master Web Interface

Kudu master processes serve their web interface on port 8051. The interface exposes several pages with information about the state of the cluster.

- A list of tablet servers, their host names, and the time of their last heartbeat.
- A list of tables, including schema and tablet location information for each.
- SQL code which you can paste into Impala Shell to add an existing table to Impala's list of known data sources.

Kudu Tablet Server Web Interface

Each tablet server serves a web interface on port 8050. The interface exposes information about each tablet hosted on the server, its current state, and debugging information about maintenance background operations.

Common Web Interface Pages

Both Kudu masters and tablet servers expose the following information via their web interfaces:

- HTTP access to server logs.
- An `/rpcz` endpoint which lists currently running RPCs via JSON.
- Details about the memory usage of different components of the process.
- The current set of configuration flags.
- Currently running threads and their resource consumption.
- A JSON endpoint exposing metrics about the server.
- The version number of the daemon deployed on the cluster.

These interfaces are linked from the landing page of each daemon's web UI.

Kudu Metrics

Kudu daemons expose a large number of metrics. Some metrics are associated with an entire server process, whereas others are associated with a particular tablet replica.

Listing available metrics

The full set of available metrics for a Kudu server can be dumped using a special command line flag:

```
$ kudu-tserver --dump_metrics_json
$ kudu-master --dump_metrics_json
```

This will output a large JSON document. Each metric indicates its name, label, description, units, and type. Because the output is JSON-formatted, this information can easily be parsed and fed into other tooling which collects metrics from Kudu servers.

If you are using Cloudera Manager, see Cloudera Manager Metrics for Kudu on page 81 for the complete list of metrics collected by Cloudera Manager for a Kudu service.

Collecting metrics via HTTP

Metrics can be collected from a server process via its HTTP interface by visiting `/metrics`. The output of this page is JSON for easy parsing by monitoring services. This endpoint accepts several GET parameters in its query string:

- `/metrics?metrics=<substring1>,<substring2>,…` - Limits the returned metrics to those which contain at least one of the provided substrings. The substrings also match entity names, so this may be used to collect metrics for a specific tablet.
- `/metrics?include_schema=1` - Includes metrics schema information such as unit, description, and label in the JSON output. This information is typically omitted to save space.
- `/metrics?compact=1` - Eliminates unnecessary whitespace from the resulting JSON, which can decrease bandwidth when fetching this page from a remote host.
- `/metrics?include_raw_histograms=1` - Include the raw buckets and values for histogram metrics, enabling accurate aggregation of percentile metrics over time and across hosts.

For example:

```
$ curl -s 'http://example-ts:8050/metrics?include_schema=1&metrics=connections_accepted'
```


```
[
    {
        "type": "server",
        "id": "kudu.tabletserver",
        "attributes": {},
        "metrics": [
            {
                "name": "rpc_connections_accepted",
                "label": "RPC Connections Accepted",
                "type": "counter",
                "unit": "connections",
                "description": "Number of incoming TCP connections made to the RPC
server",
                "value": 92
            }
        ]
```

```
    }
]
```

```
$ curl -s 'http://example-ts:8050/metrics?metrics=log_append_latency'
```


```
[
    {
        "type": "tablet",
        "id": "c0ebf9fef1b847e2a83c7bd35c2056b1",
        "attributes": {
            "table_name": "lineitem",
            "partition": "hash buckets: (55), range: [(<start>), (<end>))",
            "table_id": ""
        },
        "metrics": [
            {
                "name": "log_append_latency",
                "total_count": 7498,
                "min": 4,
                "mean": 69.3649,
                "percentile_75": 29,
                "percentile_95": 38,
                "percentile_99": 45,
                "percentile_99_9": 95,
                "percentile_99_99": 167,
                "max": 367244,
                "total_sum": 520098
            }
        ]
    }
]
```


Collecting metrics to a log

Kudu can be configured to periodically dump all of its metrics to a local log file using the `--metrics_log_interval_ms` flag. Set this flag to the interval at which metrics should be written to a log file.

The metrics log will be written to the same directory as the other Kudu log files, and with the same naming format. After any metrics log file reaches 64MB uncompressed, the log will be rolled and the previous file will be gzip-compressed.

The log file generated has three space-separated fields:

- The first field is the word `metrics`.
- The second field is the current timestamp in microseconds since the Unix epoch.
- The third is the current value of all metrics on the server, using a compact JSON encoding. The encoding is the same as the metrics fetched via HTTP described above.

> **Important:**
>
> Although metrics logging automatically rolls and compresses previous log files, it does not remove old ones. Since metrics logging can use significant amounts of disk space, consider setting up a system utility to monitor space in the log directory and archive or delete old segments.

Common Kudu workflows

Migrating to Multiple Kudu Masters

For high availability and to avoid a single point of failure, Kudu clusters should be created with multiple masters. Many Kudu clusters were created with just a single master, either for simplicity or because Kudu multi-master support was still experimental at the time. This workflow demonstrates how to migrate to a multi-master configuration.

> **Important:**
>
> - This workflow is unsafe for adding new masters to an existing multi-master configuration. Do not use it for that purpose.
> - This workflow presumes you are familiar with Kudu configuration management, with or without Cloudera Manager.
> - All of the command line steps below should be executed as the Kudu UNIX user, typically kudu.

Prepare for the migration

1. Establish a maintenance window (one hour should be sufficient). During this time the Kudu cluster will be unavailable.
2. Decide how many masters to use. The number of masters should be odd. Three or five node master configurations are recommended; they can tolerate one or two failures respectively.
3. Perform the following preparatory steps for the existing master:
   - Identify and record the directory where the master's data lives. If you are using Kudu system packages, the default value is `/var/lib/kudu/master`, but it may be customized using the `fs_wal_dir` and `fs_data_dirs` configuration parameters. If you've set `fs_data_dirs` to some directories other than the value of `fs_wal_dir`, it should be explicitly included in every command (in the following procedure) where `fs_wal_dir` is also included.
   - Identify and record the port the master is using for RPCs. The default port value is 7051, but it may have been customized using the `rpc_bind_addresses` configuration parameter.
   - Identify the master's UUID. It can be fetched using the following command:

```
$ kudu fs dump uuid --fs_wal_dir=<master_data_dir> 2>/dev/null
```

master_data_dir

The location of the existing master's previously recorded data directory.

For example:

```
$ kudu fs dump uuid --fs_wal_dir=/var/lib/kudu/master 2>/dev/null
4aab798a69e94fab8d77069edff28ce0
```

   - **(Optional)** Configure a DNS alias for the master. The alias could be a DNS cname (if the machine already has an A record in DNS), an A record (if the machine is only known by its IP address), or an alias in `/etc/hosts`. The alias should be an abstract representation of the master (e.g. `master-1`).

> **Important:**
>
> Without DNS aliases it is not possible to recover from permanent master failures, and as such it is highly recommended.

4. Perform the following preparatory steps for each new master:
   - Choose an unused machine in the cluster. The master generates very little load so it can be collocated with other data services or load-generating processes, though not with another Kudu master from the same configuration.
   - Ensure Kudu is installed on the machine, either using system packages (in which case the `kudu` and `kudu-master` packages should be installed), or some other means.

- Choose and record the directory where the master's data will live.
- Choose and record the port the master should use for RPCs.
- **(Optional)** Configure a DNS alias for the master (e.g. `master-2`, `master-3`, etc).

Perform the migration

1. Stop all the Kudu processes in the entire cluster.
2. Format the data directory on each new master machine, and record the generated UUID. Use the following commands:

```
$ kudu fs format --fs_wal_dir=<master_data_dir>
$ kudu fs dump uuid --fs_wal_dir=<master_data_dir> 2>/dev/null
```

master_data_dir

The new master's previously recorded data directory.

For example:

```
$ kudu fs format --fs_wal_dir=/var/lib/kudu/master
$ kudu fs dump uuid --fs_wal_dir=/var/lib/kudu/master 2>/dev/null
f5624e05f40649b79a757629a69d061e
```

3. If you are using Cloudera Manager, add the new Kudu master roles now, but do not start them.
   - If using DNS aliases, override the empty value of the `Master Address` parameter for each role (including the existing master role) with that master's alias.
   - Add the port number (separated by a colon) if using a non-default RPC port value.
4. Rewrite the master's Raft configuration with the following command, executed on the existing master:

```
$ kudu local_replica cmeta rewrite_raft_config --fs_wal_dir=<master_data_dir> <tablet_id>
 <all_masters>
```

master_data_dir

The existing master's previously recorded data directory

tablet_id

This must be set to the string, `00000000000000000000000000000000`.

all_masters

A space-separated list of masters, both new and existing. Each entry in the list must be a string of the form `<uuid>:<hostname>:<port>`.

uuid

The master's previously recorded UUID.

hostname

The master's previously recorded hostname or alias.

port

The master's previously recorded RPC port number.

For example:

```
$ kudu local_replica cmeta rewrite_raft_config --fs_wal_dir=/var/lib/kudu/master
00000000000000000000000000000000 4aab798a69e94fab8d77069edff28ce0:master-1:7051
```

```
f5624e05f40649b79a757629a69d061e:master-2:7051
988d8ac6530f426cbe180be5ba52033d:master-3:7051
```

5. Modify the value of the `master_addresses` configuration parameter for both existing master and new masters. The new value must be a comma-separated list of all of the masters. Each entry is a string of the form, `<hostname>:<port>`.

   hostname

   The master's previously recorded hostname or alias.

   port

   The master's previously recorded RPC port number.

6. Start the existing master.
7. Copy the master data to each new master with the following command, executed on each new master machine:

```
$ kudu local_replica copy_from_remote --fs_wal_dir=<master_data_dir> <tablet_id>
<existing_master>
```

   master_data_dir

   The new master's previously recorded data directory.

   tablet_id

   Must be set to the string, `00000000000000000000000000000000`.

   existing_master

   RPC address of the existing master. It must be a string of the form `<hostname>:<port>`.

   hostname

   The existing master's previously recorded hostname or alias.

   port

   The existing master's previously recorded RPC port number.

   Example

```
$ kudu local_replica copy_from_remote --fs_wal_dir=/var/lib/kudu/master
00000000000000000000000000000000 master-1:7051
```

8. Start all the new masters.

> **Important:** If you are using Cloudera Manager, skip the next step.

9. Modify the value of the `tserver_master_addrs` configuration parameter for each tablet server. The new value must be a comma-separated list of masters where each entry is a string of the form `<hostname>:<port>`

   hostname

   The master's previously recorded hostname or alias

   port

   The master's previously recorded RPC port number

10. Start all the tablet servers.

To verify that all masters are working properly, consider performing the following sanity checks:

- Using a browser, visit each master's web UI and navigate to the `/masters` page. All the masters should now be listed there with one master in the `LEADER` role and the others in the `FOLLOWER` role. The contents of `/masters` on each master should be the same.
- Run a Kudu system check (`ksck`) on the cluster using the `kudu` command line tool. Help for `ksck` can be viewed using the `kudu cluster ksck --help` command.

Recovering from a dead Kudu Master in a Multi-Master Deployment

Kudu multi-master deployments function normally in the event of a master loss. However, it is important to replace the dead master; otherwise a second failure may lead to a loss of availability, depending on the number of available masters. This workflow describes how to replace the dead master.

Due to KUDU-1620, it is not possible to perform this workflow without also restarting the live masters. As such, the workflow requires a maintenance window, albeit a brief one as masters generally restart quickly.

Important:

- Kudu does not yet support Raft configuration changes for masters. As such, it is only possible to replace a master if the deployment was created with DNS aliases. See the previous multi-master migration workflow for more details.
- The workflow presupposes at least basic familiarity with Kudu configuration management. If using Cloudera Manager, the workflow also presupposes familiarity with it.
- All of the command line steps below should be executed as the Kudu UNIX user, typically `kudu`.

Prepare for the recovery

1. Ensure that the dead master is well and truly dead. Take whatever steps needed to prevent it from accidentally restarting; this can be quite dangerous for the cluster post-recovery.
2. Choose one of the remaining live masters to serve as a basis for recovery. The rest of this workflow will refer to this master as the "reference" master.
3. Choose an unused machine in the cluster where the new master will live. The master generates very little load so it can be colocated with other data services or load-generating processes, though not with another Kudu master from the same configuration. The rest of this workflow will refer to this master as the "replacement" master.
4. Perform the following preparatory steps for the replacement master:
   - Ensure Kudu is installed on the machine, either via system packages (in which case the `kudu` and `kudu-master` packages should be installed), or via some other means.
   - Choose and record the directory where the master's data will live.
5. Perform the following preparatory steps for each live master:
   - Identify and record the directory where the master's data lives. If using Kudu system packages, the default value is /var/lib/kudu/master, but it may be customized via the `fs_wal_dir` and `fs_data_dirs` configuration parameter. Please note if you've set fs_data_dirs to some directories other than the value of fs_wal_dir, it should be explicitly included in every command below where fs_wal_dir is also included.
   - Identify and record the master's UUID. It can be fetched using the following command:

```
$ kudu fs dump uuid --fs_wal_dir=<master_data_dir> 2>/dev/null
```

master_data_dir

live master's previously recorded data directory

Example

```
$ kudu fs dump uuid --fs_wal_dir=/var/lib/kudu/master 2>/dev/null
80a82c4b8a9f4c819bab744927ad765c
```

6. Perform the following preparatory steps for the reference master:

   - Identify and record the directory where the master's data lives. If using Kudu system packages, the default value is `/var/lib/kudu/master`, but it may be customized using the `fs_wal_dir` and `fs_data_dirs` configuration parameter. If you have set `fs_data_dirs` to some directories other than the value of `fs_wal_dir`, it should be explicitly included in every command below where `fs_wal_dir` is also included.
   - Identify and record the UUIDs of every master in the cluster, using the following command:

```
$ kudu local_replica cmeta print_replica_uuids --fs_wal_dir=<master_data_dir> <tablet_id>
 2>/dev/null
```

master_data_dir

The reference master's previously recorded data directory.

tablet_id

Must be set to the string, `00000000000000000000000000000000`.

Example

```
$ kudu local_replica cmeta print_replica_uuids --fs_wal_dir=/var/lib/kudu/master
00000000000000000000000000000000 2>/dev/null
80a82c4b8a9f4c819bab744927ad765c 2a73eeee5d47413981d9a1c637cce170
1c3f3094256347528d02ec107466aef3
```

7. Using the two previously-recorded lists of UUIDs (one for all live masters and one for all masters), determine and record (by process of elimination) the UUID of the dead master.

Perform the recovery

1. Format the data directory on the replacement master machine using the previously recorded UUID of the dead master. Use the following command sequence:

```
$ kudu fs format --fs_wal_dir=<master_data_dir> --uuid=<uuid>
```

master_data_dir

The replacement master's previously recorded data directory.

uuid

The dead master's previously recorded UUID.

For example:

```
$ kudu fs format --fs_wal_dir=/var/lib/kudu/master --uuid=80a82c4b8a9f4c819bab744927ad765c
```

2. Copy the master data to the replacement master with the following command:

```
$ kudu local_replica copy_from_remote --fs_wal_dir=<master_data_dir> <tablet_id>
<reference_master>
```

master_data_dir

The replacement master's previously recorded data directory.

tablet_id

Must be set to the string, `00000000000000000000000000000000`.

reference_master

The RPC address of the reference master. It must be a string of the form `<hostname>:<port>`.

hostname

The reference master's previously recorded hostname or alias.

port

The reference master's previously recorded RPC port number.

For example:

```
$ kudu local_replica copy_from_remote --fs_wal_dir=/var/lib/kudu/master
00000000000000000000000000000000 master-2:7051
```

3. If you are using Cloudera Manager, add the replacement Kudu master role now, but do not start it.
   - Override the empty value of the `Master Address` parameter for the new role with the replacement master's alias.
   - If you are using a non-default RPC port, add the port number (separated by a colon) as well.
4. Reconfigure the DNS alias for the dead master to point to the replacement master.
5. Start the replacement master.
6. Restart the existing live masters. This results in a brief availability outage, but it should last only as long as it takes for the masters to come back up.

To verify that all masters are working properly, consider performing the following sanity checks:

- Using a browser, visit each master's web UI and navigate to the `/masters` page. All the masters should now be listed there with one master in the `LEADER` role and the others in the `FOLLOWER` role. The contents of `/masters` on each master should be the same.
- Run a Kudu system check (`ksck`) on the cluster using the `kudu` command line tool. Help for `ksck` can be viewed using the `kudu cluster ksck --help` command.

Developing Applications With Apache Kudu

Apache Kudu provides C++ and Java client APIs, as well as reference examples to illustrate their use. A Python API is included, but it is currently considered experimental, unstable, and is subject to change at any time.

> **Warning:** Use of server-side or private interfaces is not supported, and interfaces which are not part of public APIs have no stability guarantees.

Viewing the API Documentation

C++ API Documentation

The documentation for the C++ client APIs is included in the header files in `/usr/include/kudu/` if you installed Kudu using packages or subdirectories of `src/kudu/client/` if you built Kudu from source. If you installed Kudu using parcels, no headers are included in your installation. and you will need to build Kudu from source in order to have access to the headers and shared libraries.

The following command is a naive approach to finding relevant header files. Use of any APIs other than the client APIs is unsupported.

```
$ find /usr/include/kudu -type f -name *.h
```

Java API Documentation

View the Java API documentation online. Alternatively, after building the Java client, Java API documentation is available in `java/kudu-client/target/apidocs/index.html`.

Kudu Example Applications

Several example applications are provided in the kudu-examples Github repository. Each example includes a README that shows how to compile and run it. These examples illustrate correct usage of the Kudu APIs, as well as how to set up a virtual machine to run Kudu. The following list includes a few of the examples that are available today.

java-example

A simple Java application which connects to a Kudu instance, creates a table, writes data to it, then drops the table.

java/collectl

A simple Java application which listens on a TCP socket for time series data corresponding to the Collectl wire protocol. The commonly-available `collectl` tool can be used to send example data to the server.

`java/insert-loadgen`

A Java application that generates random insert load.

`python/dstat-kudu`

An example program that shows how to use the Kudu Python API to load data into a new / existing Kudu table generated by an external program, `dstat` in this case.

`python/graphite-kudu`

An experimental plugin for using graphite-web with Kudu as a backend.

demo-vm-setup

Scripts to download and run a VirtualBox virtual machine with Kudu already installed. For more information see the Kudu Quickstart documentation.

These examples should serve as helpful starting points for your own Kudu applications and integrations.

Maven Artifacts

he following Maven <dependency> element is valid for the Apache Kudu GA release:

```
<dependency>
  <groupId>org.apache.kudu</groupId>
  <artifactId>kudu-client</artifactId>
  <version>1.1.0</version>
</dependency>
```

Convenience binary artifacts for the Java client and various Java integrations (e.g. Spark, Flume) are also now available via the ASF Maven repository and the Central Maven repository.

Kudu Python Client

The Kudu Python client provides a Python friendly interface to the C++ client API. The sample below demonstrates the use of part of the Python client.

```
import kudu
from kudu.client import Partitioning
from datetime import datetime

# Connect to Kudu master server
client = kudu.connect(host='kudu.master', port=7051)

# Define a schema for a new table
builder = kudu.schema_builder()
builder.add_column('key').type(kudu.int64).nullable(False).primary_key()
builder.add_column('ts_val', type_=kudu.unixtime_micros, nullable=False,
compression='lz4')
schema = builder.build()

# Define partitioning schema
partitioning = Partitioning().add_hash_partitions(column_names=['key'], num_buckets=3)

# Create new table
client.create_table('python-example', schema, partitioning)

# Open a table
table = client.table('python-example')

# Create a new session so that we can apply write operations
session = client.new_session()

# Insert a row
op = table.new_insert({'key': 1, 'ts_val': datetime.utcnow()})
session.apply(op)

# Upsert a row
op = table.new_upsert({'key': 2, 'ts_val': "2016-01-01T00:00:00.000000"})
session.apply(op)

# Updating a row
op = table.new_update({'key': 1, 'ts_val': ("2017-01-01", "%Y-%m-%d")})
session.apply(op)

# Delete a row
op = table.new_delete({'key': 2})
session.apply(op)

# Flush write operations, if failures occur, capture print them.
```

```
try:
    session.flush()
except kudu.KuduBadStatus as e:
    print(session.get_pending_errors())

# Create a scanner and add a predicate
scanner = table.scanner()
scanner.add_predicate(table['ts_val'] == datetime(2017, 1, 1))

# Open Scanner and read all tuples
# Note: This doesn't scale for large scans
result = scanner.open().read_all_tuples()
```

Example Apache Impala Commands With Kudu

See Using Apache Impala (incubating) with Kudu on page 56 for guidance on installing and using Impala with Kudu, including several `impala-shell` examples.

Kudu Integration with Spark

Kudu integrates with Spark through the Data Source API as of version 1.0.0. Include the `kudu-spark` dependency using the `--packages` option:

Use the `kudu-spark_2.10` artifact if using Spark with Scala 2.10

```
spark-shell --packages org.apache.kudu:kudu-spark_2.10:1.1.0
```

Use the `kudu-spark2_2.11` artifact if using Spark 2 with Scala 2.11

```
spark-shell --packages org.apache.kudu:kudu-spark2_2.11:1.1.0
```

then import kudu-spark and create a dataframe:

```
import org.apache.kudu.spark.kudu._

// Read a table from Kudu
val df = sqlContext.read.options(Map("kudu.master" -> "kudu.master:7051","kudu.table"
-> "kudu_table")).kudu

// Query using the Spark API...
df.select("id").filter("id" >= 5).show()

// ...or register a temporary table and use SQL
df.registerTempTable("kudu_table")
val filteredDF = sqlContext.sql("select id from kudu_table where id >= 5").show()

// 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 data
kuduContext.insertRows(df, "test_table")

// Delete data
kuduContext.deleteRows(filteredDF, "test_table")

// Upsert data
kuduContext.upsertRows(df, "test_table")

// Update data
```

```
val alteredDF = df.select("id", $"count" + 1)
kuduContext.updateRows(filteredRows, "test_table"

// Data can also be inserted into the Kudu table using the data source, though the
methods on KuduContext are preferred
// NB: The default is to upsert rows; to perform standard inserts instead, set operation
 = insert in the options map
// NB: Only mode Append is supported
df.write.options(Map("kudu.master"-> "kudu.master:7051", "kudu.table"->
"test_table")).mode("append").kudu

// Check for the existence of a Kudu table
kuduContext.tableExists("another_table")

// Delete a Kudu table
kuduContext.deleteTable("unwanted_table")
```

Spark Integration Known Issues and Limitations

- Kudu tables with a name containing upper case or non-ASCII characters must be assigned an alternate name when registered as a temporary table.
- Kudu tables with a column name containing upper case or non-ASCII characters may not be used with SparkSQL. Non-primary key columns may be renamed in Kudu to work around this issue.
- `NULL`, `NOT NULL`, `<>`, `OR`, `LIKE`, and `IN` predicates are not pushed to Kudu, and instead will be evaluated by the Spark task.
- Kudu does not support all types supported by Spark SQL, such as `Date`, `Decimal` and complex types.

Integration with MapReduce, YARN, and Other Frameworks

Kudu was designed to integrate with MapReduce, YARN, Spark, and other frameworks in the Hadoop ecosystem. See RowCounter.java and ImportCsv.java for examples which you can model your own integrations on.

Using Apache Impala (incubating) with Kudu

Apache Kudu has tight integration with Apache Impala (incubating), allowing you to use Impala to insert, query, update, and delete data from Kudu tablets using Impala's SQL syntax, as an alternative to using the Kudu APIs to build a custom Kudu application. In addition, you can use JDBC or ODBC to connect existing or new applications written in any language, framework, or business intelligence tool to your Kudu data, using Impala as the broker.

Prerequisites

- To use Impala to query Kudu data as described in this topic, you will require Cloudera Manager 5.10.x and CDH 5.10.x or later.
- The syntax described in this topic is specific to Impala 2.8 (ships with CDH 5.10) and above, and will not work on previous versions. If you are using an earlier version of Impala (including the `IMPALA_KUDU` releases previously available), upgrade to Impala 2.8.

  Note that this topic does not describe Impala installation or upgrade procedures. Refer to the Impala documentation to make sure you are able to run queries against Impala tables on HDFS before proceeding.
- Lower versions of CDH and Cloudera Manager used an experimental fork of Impala which is referred to as `IMPALA_KUDU`. If you have previously installed the `IMPALA_KUDU` service, make sure you remove it from your cluster before you proceed. Install Kudu 1.2.x (or later) using either Cloudera Manager or the command-line.

Impala Database Containment Model

Every Impala table is contained within a namespace called a database. The default database is called `default`, and you may create and drop additional databases as desired. To create the database, use a `CREATE DATABASE` statement. To use the database for further Impala operations such as `CREATE TABLE`, use the USE statement. For example, to create a table in a database called *impala_kudu*, use the following statements:

```
CREATE DATABASE impala_kudu;
USE impala_kudu;
CREATE TABLE my_first_table (
...
```

The `my_first_table` table is created within the `impala_kudu` database. To refer to this database in the future, without using a specific USE statement, you can refer to the table using `<database>:<table>` syntax. For example, to specify the `my_first_table` table in database `impala_kudu`, as opposed to any other table with the same name in another database, refer to the table as `impala_kudu:my_first_table`. This also applies to `INSERT`, `UPDATE`, `DELETE`, and `DROP` statements.

> **Warning:** Currently, Kudu does not encode the Impala database into the table name in any way. This means that even though you can create Kudu tables within Impala databases, the actual Kudu tables need to be unique within Kudu. For example, if you create `database_1:my_kudu_table` and `database_2:my_kudu_table`, you will have a naming collision within Kudu, even though this would not cause a problem in Impala.

Internal and External Impala Tables

When creating a new Kudu table using Impala, you can create the table as an internal table or an external table.

Internal

An internal table (created by `CREATE TABLE`) is managed by Impala, and can be dropped by Impala. When you create a new table using Impala, it is generally a internal table. When such a table is created in Impala, the corresponding Kudu table will be named `my_database::table_name`.

External

An external table (created by `CREATE EXTERNAL TABLE`) is not managed by Impala, and dropping such a table does not drop the table from its source location (here, Kudu). Instead, it only removes the mapping between Impala and Kudu. This is the mode used in the syntax provided by Kudu for mapping an existing table to Impala.

See the Impala documentation for more information about internal and external tables.

Using Impala To Query Kudu Tables

Neither Kudu nor Impala need special configuration in order for you to use the Impala Shell or the Impala API to insert, update, delete, or query Kudu data using Impala. However, you do need to create a mapping between the Impala and Kudu tables. Kudu provides the Impala query to map to an existing Kudu table in the web UI.

- Make sure you are using the `impala-shell` binary provided by the default CDH Impala binary. The following example shows how you can verify this using the `alternatives` command on a RHEL 6 host. Do not copy and paste the `alternatives --set` command directly, because the file names are likely to differ.

```
$ sudo alternatives --display impala-shell

impala-shell - status is auto.
 link currently points to
/opt/cloudera/parcels/CDH-5.10.0-1.cdh5.10.0.p0.25/bin/impala-shell
/opt/cloudera/parcels/CDH-5.10.0-1.cdh5.10.0.p0.25/bin/impala-shell - priority 10
Current `best' version is
/opt/cloudera/parcels/CDH-5.10.0-1.cdh5.10.0.p0.25/bin/impala-shell.
```

- Although not necessary, it is recommended that you configure Impala with the locations of the Kudu Masters using the `--kudu_master_hosts=<master1>[:port]` flag. If this flag is not set, you will need to manually provide this configuration each time you create a table by specifying the `kudu_master_addresses` property inside a TBLPROPERTIES clause. If you are using Cloudera Manager, no such configuration is needed. The Impala service will automatically recognize the Kudu Master hosts.

  The rest of this guide assumes that this configuration has been set.

- Start Impala Shell using the `impala-shell` command. By default, `impala-shell` attempts to connect to the Impala daemon on `localhost` on port 21000. To connect to a different host, use the `-i <host:port>` option.

  To automatically connect to a specific Impala database, use the `-d <database>` option. For instance, if all your Kudu tables are in Impala in the database `impala_kudu`, use `-d impala_kudu` to use this database.

- To quit the Impala Shell, use the following command: `quit;`

Querying an Existing Kudu Table from Impala

Tables created through the Kudu API or other integrations such as Apache Spark are not automatically visible in Impala. To query them, you must first create an external table within Impala to map the Kudu table into an Impala database:

```
CREATE EXTERNAL TABLE my_mapping_table
STORED AS KUDU
TBLPROPERTIES (
  'kudu.table_name' = 'my_kudu_table'
);
```

Creating a New Kudu Table From Impala

Creating a new table in Kudu from Impala is similar to mapping an existing Kudu table to an Impala table, except that you need to specify the schema and partitioning information yourself. Use the following example as a guideline. Impala first creates the table, then creates the mapping.

In the `CREATE TABLE` statement, the columns that comprise the primary key must be listed first. Additionally, primary key columns are implicitly considered `NOT NULL`.

When creating a new table in Kudu, you *must define a partition schema* to pre-split your table. The best partition schema to use depends upon the structure of your data and your data access patterns. The goal is to maximize parallelism and use all your tablet servers evenly. For more information on partition schemas, see Partitioning Tables on page 58.

The following `CREATE TABLE` example distributes the table into 16 partitions by hashing the `id` column, for simplicity.

```
CREATE TABLE my_first_table
(
  id BIGINT,
  name STRING,
  PRIMARY KEY(id)
)
PARTITION BY HASH PARTITIONS 16
STORED AS KUDU;
```

CREATE TABLE AS SELECT

You can create a table by querying any other table or tables in Impala, using a `CREATE TABLE ... AS SELECT` statement. The following example imports all rows from an existing table, `old_table`, into a new Kudu table, `new_table`. The columns in `new_table` will have the same names and types as the columns in `old_table`, but you will need to additionally specify the primary key and partitioning schema.

```
CREATE TABLE new_table
PRIMARY KEY (ts, name)
PARTITION BY HASH(name) PARTITIONS 8
STORED AS KUDU
AS SELECT ts, name, value FROM old_table;
```

You can refine the `SELECT` statement to only match the rows and columns you want to be inserted into the new table. You can also rename the columns by using syntax like `SELECT name as new_col_name`.

Partitioning Tables

Tables are partitioned into tablets according to a partition schema on the primary key columns. Each tablet is served by at least one tablet server. Ideally, a table should be split into tablets that are distributed across a number of tablet servers to maximize parallel operations. The details of the partitioning schema you use will depend entirely on the type of data you store and how you access it.

Kudu currently has no mechanism for splitting or merging tablets after the table has been created. Until this feature has been implemented, you must provide a partition schema for your table when you create it. When designing your tables, consider using primary keys that will allow you to partition your table into tablets which grow at similar rates

You can partition your table using Impala's `PARTITION BY` clause, which supports distribution by RANGE or HASH. The partition scheme can contain zero or more HASH definitions, followed by an optional RANGE definition. The RANGE definition can refer to one or more primary key columns. Examples of basic and advanced partitioning are shown below.

Monotonically Increasing Values - If you partition by range on a column whose values are monotonically increasing, the last tablet will grow much larger than the others. Additionally, all data being inserted will be written to a single tablet at a time, limiting the scalability of data ingest. In that case, consider distributing by HASH instead of, or in addition to, RANGE.

> **Note:** Impala keywords, such as `group`, are enclosed by back-tick characters when they are used as identifiers, rather than as keywords.

Basic Partitioning

PARTITION BY RANGE

You can specify range partitions for one or more primary key columns. Range partitioning in Kudu allows splitting a table based based on specific values or ranges of values of the chosen partition keys. This allows you to balance parallelism in writes with scan efficiency.

For instance, if you have a table that has the columns `state`, `name`, and `purchase_count`, and you partition the table by `state`, it will create 50 tablets, one for each US state.

```
CREATE TABLE customers (
  state STRING,
  name STRING,
  purchase_count int,
  PRIMARY KEY (state, name)
)
PARTITION BY RANGE (state)
(
  PARTITION VALUE = 'al',
  PARTITION VALUE = 'ak',
  PARTITION VALUE = 'ar',
  ...
  ...
  PARTITION VALUE = 'wv',
  PARTITION VALUE = 'wy'
)
STORED AS KUDU;
```

PARTITION BY HASH

Instead of distributing by an explicit range, or in combination with range distribution, you can distribute into a specific number of partitions by hash. You specify the primary key columns you want to partition by, and the number of partitions you want to use. Rows are distributed by hashing the specified key columns. Assuming that the values being hashed do not themselves exhibit significant skew, this will serve to distribute the data evenly across all partitions.

You can specify multiple definitions, and you can specify definitions which use compound primary keys. However, one column cannot be mentioned in multiple hash definitions. Consider two columns, a and b:

- `HASH(a), HASH(b)` -- will succeed
- `HASH(a,b)` -- will succeed
- `HASH(a), HASH(a,b)` -- will fail

> **Note:** `PARTITION BY HASH` with no column specified is a shortcut to create the desired number of partitions by hashing all primary key columns.

Hash partitioning is a reasonable approach if primary key values are evenly distributed in their domain and no data skew is apparent, such as timestamps or serial IDs.

The following example creates 16 tablets by hashing the `id` column. A maximum of 16 tablets can be written to in parallel. In this example, a query for a range of `sku` values is likely to need to read from all 16 tablets, so this may not be the optimum schema for this table. See Advanced Partitioning on page 60 for an extended example.

```
CREATE TABLE cust_behavior (
  id BIGINT,
  sku STRING,
  salary STRING,
  edu_level INT,
  usergender STRING,
```

```
  `group` STRING,
  city STRING,
  postcode STRING,
  last_purchase_price FLOAT,
  last_purchase_date BIGINT,
  category STRING,
  rating INT,
  fulfilled_date BIGINT,
  PRIMARY KEY (id, sku)
)
PARTITION BY HASH PARTITIONS 16
STORED AS KUDU;
```

Advanced Partitioning

You can combine HASH and RANGE partitioning to create more complex partition schemas. You can also specify zero or more HASH definitions, followed by zero or one RANGE definitions. Each schema definition can encompass one or more columns. While enumerating every possible distribution schema is out of the scope of this topic, the following examples illustrate some of the possibilities.

PARTITION BY HASH and RANGE

Consider the basic `PARTITION BY HASH` example above. If you often query for a range of `sku` values, you can optimize the example by combining hash partitioning with range partitioning.

The following example still creates 16 tablets, by first hashing the id column into 4 partitions, and then applying range partitioning to split each partition into four tablets, based upon the value of the `sku` string. At least four tablets (and possibly up to 16) can be written to in parallel, and when you query for a contiguous range of `sku` values, there's a good chance you only need to read a quarter of the tablets to fulfill the query.

By default, the entire primary key `(id, sku)` will be hashed when you use `PARTITION BY HASH`. To hash on only part of the primary key, and use a range partition on the rest, use the syntax demonstrated below.

```
CREATE TABLE cust_behavior (
  id BIGINT,
  sku STRING,
  salary STRING,
  edu_level INT,
  usergender STRING,
  `group` STRING,
  city STRING,
  postcode STRING,
  last_purchase_price FLOAT,
  last_purchase_date BIGINT,
  category STRING,
  rating INT,
  fulfilled_date BIGINT,
  PRIMARY KEY (id, sku)
)
PARTITION BY HASH (id) PARTITIONS 4,
RANGE (sku)
(
  PARTITION VALUES < 'g',
  PARTITION 'g' <= VALUES < 'o',
  PARTITION 'o' <= VALUES < 'u',
  PARTITION 'u' <= VALUES
)
STORED AS KUDU;
```

Multiple PARTITION BY HASH Definitions

Once again expanding on the example above, let's assume that the pattern of incoming queries will be unpredictable, but you still want to ensure that writes are spread across a large number of tablets. You can achieve maximum distribution across the entire primary key by hashing on both primary key columns.

```
CREATE TABLE cust_behavior (
  id BIGINT,
```

```
  sku STRING,
  salary STRING,
  edu_level INT,
  usergender STRING,
  `group` STRING,
  city STRING,
  postcode STRING,
  last_purchase_price FLOAT,
  last_purchase_date BIGINT,
  category STRING,
  rating INT,
  fulfilled_date BIGINT,
  PRIMARY KEY (id, sku)
)
PARTITION BY HASH (id) PARTITIONS 4,
             HASH (sku) PARTITIONS 4
STORED AS KUDU;
```

The example creates 16 partitions. You could also use `HASH (id, sku) PARTITIONS 16`. However, a scan for `sku` values would almost always impact all 16 partitions, rather than possibly being limited to 4.

Non-Covering Range Partitions

Kudu 1.x (and higher) supports the use of non-covering range partitions, which can be used to address the following scenarios:

- In the case of time-series data or other schemas which need to account for constantly-increasing primary keys, tablets serving old data will be relatively fixed in size, while tablets receiving new data will grow without bounds.
- In cases where you want to partition data based on its category, such as sales region or product type, without non-covering range partitions you must know all of the partitions ahead of time or manually recreate your table if partitions need to be added or removed, such as the introduction or elimination of a product type.

> **Note:** Non-covering range partitions have some caveats. Be sure to read the link:/docs/schema_design.html [Schema Design guide].

The following example creates a tablet per year (5 tablets total), for storing log data. The table only accepts data from 2012 to 2016. Keys outside of these ranges will be rejected.

```
CREATE TABLE sales_by_year (
  year INT, sale_id INT, amount INT,
  PRIMARY KEY (sale_id, year)
)
PARTITION BY RANGE (year) (
  PARTITION VALUE = 2012,
  PARTITION VALUE = 2013,
  PARTITION VALUE = 2014,
  PARTITION VALUE = 2015,
  PARTITION VALUE = 2016
)
STORED AS KUDU;
```

When records start coming in for 2017, they will be rejected. At that point, the `2017` range should be added as follows:

```
ALTER TABLE sales_by_year ADD RANGE PARTITION VALUE = 2017;
```

In use cases where a rolling window of data retention is required, range partitions may also be dropped. For example, if data from 2012 should no longer be retained, it may be deleted in bulk:

```
ALTER TABLE sales_by_year DROP RANGE PARTITION VALUE = 2012;
```

Note that just like dropping a table, this irrecoverably deletes all data stored in the dropped partition.

Partitioning Guidelines

- For large tables, such as fact tables, aim for as many tablets as you have cores in the cluster.
- For small tables, such as dimension tables, aim for a large enough number of tablets that each tablet is at least 1 GB in size.

In general, be mindful the number of tablets limits the parallelism of reads, in the current implementation. Increasing the number of tablets significantly beyond the number of cores is likely to have diminishing returns.

Optimizing Performance for Evaluating SQL Predicates

If the `WHERE` clause of your query includes comparisons with the operators =, <=, <, >, >=, `BETWEEN`, or `IN`, Kudu evaluates the condition directly and only returns the relevant results. This provides optimum performance, because Kudu only returns the relevant results to Impala.

For predicates such as `!=`, `LIKE`, or any other predicate type supported by Impala, Kudu does not evaluate the predicates directly. Instead, it returns all results to Impala and relies on Impala to evaluate the remaining predicates and filter the results accordingly. This may cause differences in performance, depending on the delta of the result set before and after evaluating the `WHERE` clause. In some cases, creating and periodically updating materialized views may be the right solution to work around these inefficiencies.

Inserting a Row

The syntax for inserting one or more rows using Impala is shown below.

```
INSERT INTO my_first_table VALUES (99, "sarah");
INSERT INTO my_first_table VALUES (1, "john"), (2, "jane"), (3, "jim");
```

The primary key must not be null.

Inserting In Bulk

When inserting in bulk, there are at least three common choices. Each may have advantages and disadvantages, depending on your data and circumstances.

Multiple Single INSERT statements

This approach has the advantage of being easy to understand and implement. This approach is likely to be inefficient because Impala has a high query start-up cost compared to Kudu's insertion performance. This will lead to relatively high latency and poor throughput.

Single INSERT statement with multiple VALUES subclauses

If you include more than 1024 `VALUES` statements, Impala batches them into groups of 1024 (or the value of `batch_size`) before sending the requests to Kudu. This approach may perform slightly better than multiple sequential `INSERT` statements by amortizing the query start-up penalties on the Impala side. To set the batch size for the current Impala Shell session, use the following syntax:

```
set batch_size=10000;
```

> **Note:** Increasing the Impala batch size causes Impala to use more memory. You should verify the impact on your cluster and tune accordingly.

Batch Insert

The approach that usually performs best, from the standpoint of both Impala and Kudu, is usually to import the data using a `SELECT FROM` subclause in Impala.

1. If your data is not already in Impala, one strategy is to import it from a text file, such as a TSV or CSV file.
2. Create the Kudu table, being mindful that the columns designated as primary keys cannot have null values.

3. Insert values into the Kudu table by querying the table containing the original data, as in the following example:

```
INSERT INTO my_kudu_table
SELECT * FROM legacy_data_import_table;
```

Ingest using the C++ or Java API

In many cases, the appropriate ingest path is to use the C++ or Java API to insert directly into Kudu tables. Unlike other Impala tables, data inserted into Kudu tables using the API becomes available for query in Impala without the need for any `INVALIDATE METADATA` statements or other statements needed for other Impala storage types.

INSERT and Primary Key Uniqueness Violations

In many relational databases, if you try to insert a row that has already been inserted, the insertion will fail because the primary key will be duplicated (see Failures During INSERT, UPDATE, UPSERT, and DELETE Operations on page 64). Impala, however, does not fail the query. Instead, it will generate a warning and continue to execute the remainder of the insert statement.

If you meant to replace existing rows from the table, use the UPSERT statement instead.

```
INSERT INTO my_first_table VALUES (99, "sarah");
UPSERT INTO my_first_table VALUES (99, "zoe");
```

The current value of the row is now `zoe`.

Updating a Row

The syntax for updating one or more rows using Impala is shown below.

```
UPDATE my_first_table SET name="bob" where id = 3;
```

You cannot change or null the primary key value.

> **Important:** The UPDATE statement only works in Impala when the underlying data source is Kudu.

Updating In Bulk

You can update in bulk using the same approaches outlined in Inserting In Bulk on page 62.

Upserting a Row

The UPSERT command acts as a combination of the `INSERT` and UPDATE statements. For each row processed by the UPSERT statement:

- If another row already exists with the same set of primary key values, the other columns are updated to match the values from the row being “UPSERTed”.
- If there is no row with the same set of primary key values, the row is created, the same as if the `INSERT` statement was used.

```
UPSERT INTO my_first_table VALUES (1, "jonathan"), (4, "james");
```

Deleting a Row

You can delete Kudu rows in near real time using Impala.

```
DELETE FROM my_first_table WHERE id < 3;
```

You can even use more complex joins when deleting rows. For example, Impala uses a comma in the FROM sub-clause to specify a join query.

```
DELETE c FROM my_second_table c, stock_symbols s WHERE c.name = s.symbol;
```

> **Important:** The DELETE statement only works in Impala when the underlying data source is Kudu.

Deleting In Bulk

You can delete in bulk using the same approaches outlined in Inserting In Bulk on page 62.

Failures During INSERT, UPDATE, UPSERT, and DELETE Operations

INSERT, UPDATE, and DELETE statements cannot be considered transactional as a whole. If one of these operations fails part of the way through, the keys may have already been created (in the case of INSERT) or the records may have already been modified or removed by another process (in the case of UPDATE or DELETE). You should design your application with this in mind.

Altering Table Properties

You can change Impala's metadata relating to a given Kudu table by altering the table's properties. These properties include the table name, the list of Kudu master addresses, and whether the table is managed by Impala (internal) or externally. You cannot modify a table's split rows after table creation.

> **Important:** Altering table properties only changes Impala's metadata about the table, not the underlying table itself. These statements do not modify any Kudu data.

Rename an Impala Mapping Table

```
ALTER TABLE my_table RENAME TO my_new_table;
```

Renaming a table using the ALTER TABLE ... RENAME statement only renames the Impala mapping table, regardless of whether the table is an internal or external table. This avoids disruption to other applications that may be accessing the underlying Kudu table.

Rename the underlying Kudu table for an internal table

If a table is an internal table, the underlying Kudu table may be renamed by changing the kudu.table_name property:

```
ALTER TABLE my_internal_table
SET TBLPROPERTIES('kudu.table_name' = 'new_name')
```

Remapping an external table to a different Kudu table

If another application has renamed a Kudu table under Impala, it is possible to re-map an external table to point to a different Kudu table name.

```
ALTER TABLE my_external_table_
SET TBLPROPERTIES('kudu.table_name' = 'some_other_kudu_table')
```

Change the Kudu Master Addresses

```
ALTER TABLE my_table SET TBLPROPERTIES('kudu.master_addresses' =
'kudu-original-master.example.com:7051,kudu-new-master.example.com:7051');
```

Change an Internally-Managed Table to External

```
ALTER TABLE my_table SET TBLPROPERTIES('EXTERNAL' = 'TRUE');
```

Dropping a Kudu Table using Impala

If the table was created as an internal table in Impala, using `CREATE TABLE`, the standard `DROP TABLE` syntax drops the underlying Kudu table and all its data. If the table was created as an external table, using `CREATE EXTERNAL TABLE`, the mapping between Impala and Kudu is dropped, but the Kudu table is left intact, with all its data. To change an external table to internal, or vice versa, see Altering Table Properties on page 64.

```
DROP TABLE my_first_table;
```

Known Issues and Limitations

- When creating a Kudu table, the `CREATE TABLE` statement must include the primary key columns before other columns, in primary key order.
- Impala cannot update values in primary key columns.
- Impala cannot create Kudu tables with `TIMESTAMP`, `DECIMAL`, `VARCHAR`, or nested-typed columns.
- Kudu tables with a name containing upper case or non-ASCII characters must be assigned an alternate name when used as an external table in Impala.
- Kudu tables with a column name containing upper case or non-ASCII characters may not be used as an external table in Impala. Non-primary key columns may be renamed in Kudu to work around this issue.
- Kudu tables containing `UNIXTIME_MICROS`-typed columns may not be used as an external table in Impala.
- `NULL`, `NOT NULL`, `!=`, and `LIKE` predicates are not pushed to Kudu, and instead will be evaluated by the Impala scan node. This may decrease performance relative to other types of predicates.
- Updates, inserts, and deletes using Impala are non-transactional. If a query fails part of the way through, its partial effects will not be rolled back.
- The maximum parallelism of a single query is limited to the number of tablets in a table. For good analytic performance, aim for 10 or more tablets per host or use large tables.

Impala Keywords Not Supported for Creating Kudu Tables

- `PARTITIONED`
- `LOCATION`
- `ROWFORMAT`

Next Steps

The examples above have only explored a fraction of what you can do with Impala Shell.

- Learn about the Impala project.
- Read the Impala documentation.
- View the Impala SQL Reference.
- For in-depth information on how to configure and use Impala to query Kudu data, see Integrating Impala with Kudu.
- Read about Impala internals or learn how to contribute to Impala on the Impala Wiki.

Apache Kudu Schema Design

Kudu tables have a structured data model similar to tables in a traditional relational database. With Kudu, schema design is critical for achieving the best performance and operational stability. Every workload is unique, and there is no single schema design that is best for every table. This topic outlines effective schema design philosophies for Kudu, and how they differ from approaches used for traditional relational database schemas.

There are three main concerns when creating Kudu tables: column design, primary key design, and partitioning.

The Perfect Schema

The perfect schema would accomplish the following:

- Data would be distributed such that reads and writes are spread evenly across tablet servers. This can be achieved by effective partitioning.
- Tablets would grow at an even, predictable rate, and load across tablets would remain steady over time. This can be achieved by effective partitioning.
- Scans would read the minimum amount of data necessary to fulfill a query. This is impacted mostly by primary key design, but partitioning also plays a role via partition pruning.

The perfect schema depends on the characteristics of your data, what you need to do with it, and the topology of your cluster. Schema design is the single most important thing within your control to maximize the performance of your Kudu cluster.

Column Design

A Kudu table consists of one or more columns, each with a defined type. Columns that are not part of the primary key may be nullable. Supported column types include:

- boolean
- 8-bit signed integer
- 16-bit signed integer
- 32-bit signed integer
- 64-bit signed integer
- unixtime_micros (64-bit microseconds since the Unix epoch)
- single-precision (32-bit) IEEE-754 floating-point number
- double-precision (64-bit) IEEE-754 floating-point number
- UTF-8 encoded string (up to 64KB)
- binary (up to 64KB)

Kudu takes advantage of strongly-typed columns and a columnar on-disk storage format to provide efficient encoding and serialization. To make the most of these features, columns should be specified as the appropriate type, rather than simulating a 'schemaless' table using string or binary columns for data which could otherwise be structured. In addition to encoding, Kudu allows compression to be specified on a per-column basis.

Column Encoding

Depending on the type of the column, Kudu columns can be created with the following encoding types.

Plain Encoding

Data is stored in its natural format. For example, `int32` values are stored as fixed-size 32-bit little-endian integers.

Bitshuffle Encoding

A block of values is rearranged to store the most significant bit of every value, followed by the second most significant bit of every value, and so on. Finally, the result is LZ4 compressed. Bitshuffle encoding is a good choice for columns that have many repeated values, or values that change by small amounts when sorted by primary key. The bitshuffle project has a good overview of performance and use cases.

Run Length Encoding

Runs (consecutive repeated values) are compressed in a column by storing only the value and the count. Run length encoding is effective for columns with many consecutive repeated values when sorted by primary key.

Dictionary Encoding

A dictionary of unique values is built, and each column value is encoded as its corresponding index in the dictionary. Dictionary encoding is effective for columns with low cardinality. If the column values of a given row set are unable to be compressed because the number of unique values is too high, Kudu will transparently fall back to plain encoding for that row set. This is evaluated during flush.

Prefix Encoding

Common prefixes are compressed in consecutive column values. Prefix encoding can be effective for values that share common prefixes, or the first column of the primary key, since rows are sorted by primary key within tablets.

Each column in a Kudu table can be created with an encoding, based on the type of the column. Columns use plain encoding by default.

Column Type	Encoding
`int8, int16, int32`	`plain, bitshuffle, run length`
`int64, unixtime_micros`	`plain, bitshuffle`
`float, double`	`plain, bitshuffle`
`bool`	`plain, run length`
`string, binary`	`plain, prefix, dictionary`

Column Compression

Kudu allows per-column compression using the `LZ4`, `Snappy`, or `zlib` compression codecs. By default, columns are stored uncompressed. Consider using compression if reducing storage space is more important than raw scan performance.

Every data set will compress differently, but in general LZ4 is the most efficient codec, while `zlib` will compress to the smallest data sizes. Bitshuffle-encoded columns are automatically compressed using LZ4, so it is not recommended to apply additional compression on top of this encoding.

Primary Key Design

Every Kudu table must declare a primary key index comprised of one or more columns. Primary key columns must be non-nullable, and may not be a boolean or floating-point type. Once set during table creation, the set of columns in the primary key may not be altered. Like an RDBMS primary key, the Kudu primary key enforces a uniqueness constraint; attempting to insert a row with the same primary key values as an existing row will result in a duplicate key error.

Unlike an RDBMS, Kudu does not provide an auto-incrementing column feature, so the application must always provide the full primary key during insert. Row delete and update operations must also specify the full primary key of the row to be changed; Kudu does not natively support range deletes or updates. The primary key values of a column may not be updated after the row is inserted; however, the row may be deleted and re-inserted with the updated value.

Primary Key Index

As with many traditional relational databases, Kudu's primary key is a clustered index. All rows within a tablet are kept in primary key sorted order. Kudu scans which specify equality or range constraints on the primary key will automatically skip rows which can not satisfy the predicate. This allows individual rows to be efficiently found by specifying equality constraints on the primary key columns.

Partitioning

In order to provide scalability, Kudu tables are partitioned into units called tablets, and distributed across many tablet servers. A row always belongs to a single tablet. The method of assigning rows to tablets is determined by the partitioning of the table, which is set during table creation.

Choosing a partitioning strategy requires understanding the data model and the expected workload of a table. For write-heavy workloads, it is important to design the partitioning such that writes are spread across tablets in order to avoid overloading a single tablet. For workloads involving many short scans, where the overhead of contacting remote servers dominates, performance can be improved if all of the data for the scan is located on the same tablet. Understanding these fundamental trade-offs is central to designing an effective partition schema.

> **Important:** Kudu does not provide a default partitioning strategy when creating tables. It is recommended that new tables which are expected to have heavy read and write workloads have at least as many tablets as tablet servers.

Kudu provides two types of partitioning: range partitioning and hash partitioning. Tables may also have multilevel partitioning, which combines range and hash partitioning, or multiple instances of hash partitioning.

Range Partitioning

Range partitioning distributes rows using a totally-ordered range partition key. Each partition is assigned a contiguous segment of the range partition keyspace. The key must be comprised of a subset of the primary key columns. If the range partition columns match the primary key columns, then the range partition key of a row will equal its primary key. In range partitioned tables without hash partitioning, each range partition will correspond to exactly one tablet.

The initial set of range partitions is specified during table creation as a set of partition bounds and split rows. For each bound, a range partition will be created in the table. Each split will divide a range partition in two. If no partition bounds are specified, then the table will default to a single partition covering the entire key space (unbounded below and above). Range partitions must always be non-overlapping, and split rows must fall within a range partition.

Adding and Removing Range Partitions

Kudu allows range partitions to be dynamically added and removed from a table at runtime, without affecting the availability of other partitions. Removing a partition will delete the tablets belonging to the partition, as well as the data contained in them. Subsequent inserts into the dropped partition will fail. New partitions can be added, but they must not overlap with any existing range partitions. Kudu allows dropping and adding any number of range partitions in a single transactional alter table operation.

Dynamically adding and dropping range partitions is particularly useful for time series use cases. As time goes on, range partitions can be added to cover upcoming time ranges. For example, a table storing an event log could add a month-wide partition just before the start of each month in order to hold the upcoming events. Old range partitions can be dropped in order to efficiently remove historical data, as necessary.

Hash Partitioning

Hash partitioning distributes rows by hash value into one of many buckets. In single-level hash partitioned tables, each bucket will correspond to exactly one tablet. The number of buckets is set during table creation. Typically the primary key columns are used as the columns to hash, but as with range partitioning, any subset of the primary key columns can be used.

Hash partitioning is an effective strategy when ordered access to the table is not needed. Hash partitioning is effective for spreading writes randomly among tablets, which helps mitigate hot-spotting and uneven tablet sizes.

Multilevel Partitioning

Kudu allows a table to combine multiple levels of partitioning on a single table. Zero or more hash partition levels can be combined with an optional range partition level. The only additional constraint on multilevel partitioning beyond the constraints of the individual partition types, is that multiple levels of hash partitions must not hash the same columns.

When used correctly, multilevel partitioning can retain the benefits of the individual partitioning types, while reducing the downsides of each. The total number of tablets in a multilevel partitioned table is the product of the number of partitions in each level.

Partition Pruning

Kudu scans will automatically skip scanning entire partitions when it can be determined that the partition can be entirely filtered by the scan predicates. To prune hash partitions, the scan must include equality predicates on every hashed column. To prune range partitions, the scan must include equality or range predicates on the range partitioned columns. Scans on multilevel partitioned tables can take advantage of partition pruning on any of the levels independently.

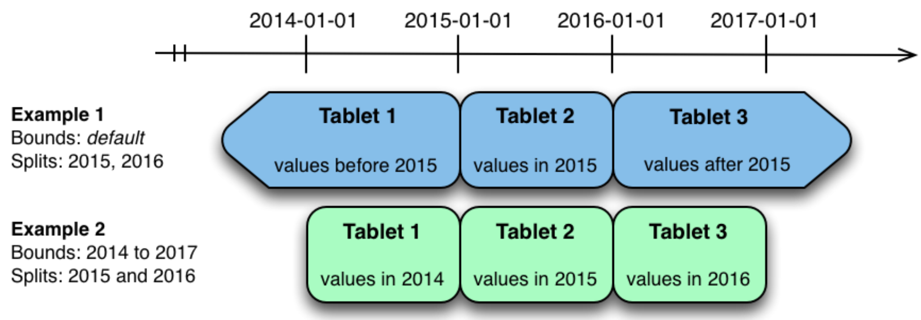
Partitioning Examples

To illustrate the factors and tradeoffs associated with designing a partitioning strategy for a table, we will walk through some different partitioning scenarios. Consider the following table schema for storing machine metrics data (using SQL syntax and date-formatted timestamps for clarity):

```
CREATE TABLE metrics (
    host STRING NOT NULL,
    metric STRING NOT NULL,
    time INT64 NOT NULL,
    value DOUBLE NOT NULL,
    PRIMARY KEY (host, metric, time),
);
```

Range Partitioning

A natural way to partition the `metrics` table is to range partition on the `time` column. Let's assume that we want to have a partition per year, and the table will hold data for 2014, 2015, and 2016. There are at least two ways that the table could be partitioned: with unbounded range partitions, or with bounded range partitions.



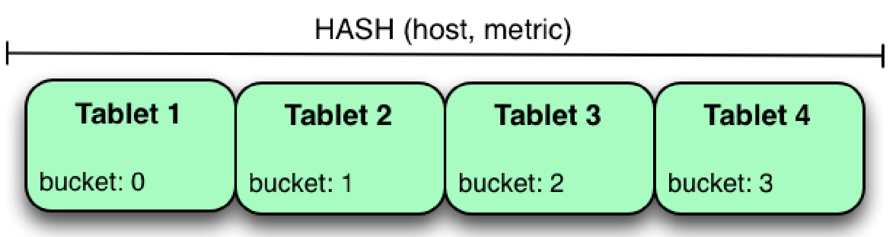
The image above shows the two ways the `metrics` table can be range partitioned on the `time` column. In the first example (in blue), the default range partition bounds are used, with splits at `2015-01-01` and `2016-01-01`. This results in three tablets: the first containing values before 2015, the second containing values in the year 2015, and the third containing values after 2016. The second example (in green) uses a range partition bound of `[(2014-01-01), (2017-01-01)]`, and splits at `2015-01-01` and `2016-01-01`. The second example could have equivalently been expressed through range partition bounds of `[(2014-01-01), (2015-01-01)]`, `[(2015-01-01), (2016-01-01)]`, and `[(2016-01-01), (2017-01-01)]`, with no splits. The first example has unbounded lower and upper range partitions, while the second example includes bounds.

Each of the range partition examples above allows time-bounded scans to prune partitions falling outside of the scan's time bound. This can greatly improve performance when there are many partitions. When writing, both examples suffer from potential hot-spotting issues. Because metrics tend to always be written at the current time, most writes will go into a single range partition.

The second example is more flexible, because it allows range partitions for future years to be added to the table. In the first example, all writes for times after `2016-01-01` will fall into the last partition, so the partition may eventually become too large for a single tablet server to handle.

Hash Partitioning

Another way of partitioning the `metrics` table is to hash partition on the `host` and `metric` columns.



In the example above, the `metrics` table is hash partitioned on the `host` and `metric` columns into four buckets. Unlike the range partitioning example earlier, this partitioning strategy will spread writes over all tablets in the table evenly, which helps overall write throughput. Scans over a specific host and metric can take advantage of partition pruning by specifying equality predicates, reducing the number of scanned tablets to one. One issue to be careful of with a pure hash partitioning strategy, is that tablets could grow indefinitely as more and more data is inserted into the table. Eventually tablets will become too big for an individual tablet server to hold.

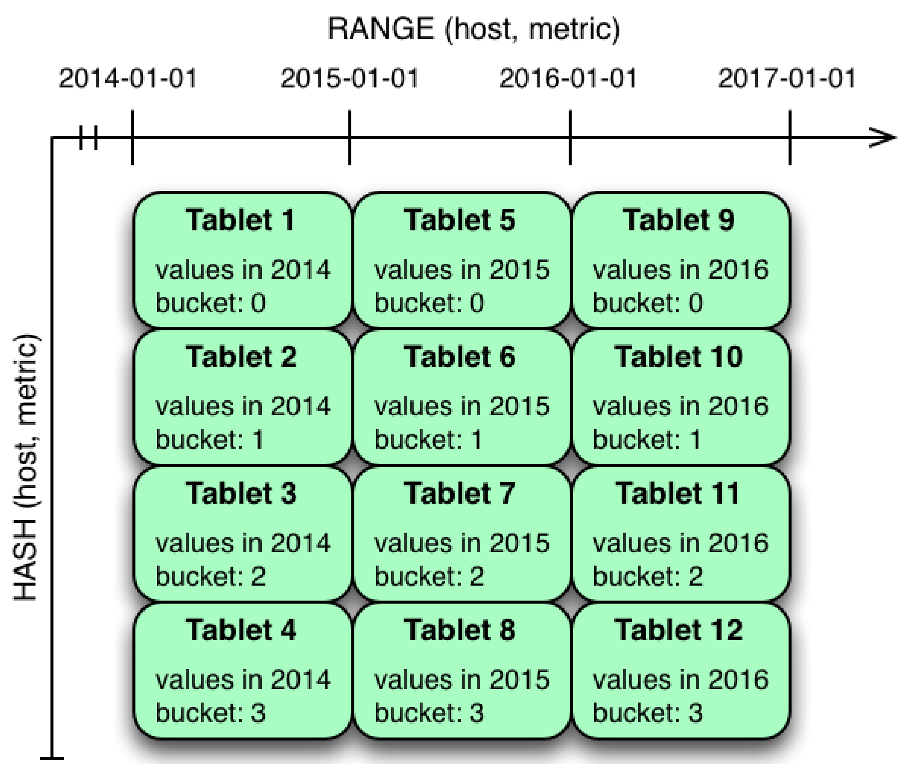
Hash and Range Partitioning

The previous examples showed how the `metrics` table could be range partitioned on the `time` column, or hash partitioned on the `host` and `metric` columns. These strategies have associated strength and weaknesses:

Table 4: Partitioning Strategies

Strategy	Writes	Reads	Tablet Growth
`range(time)`	⍰ all writes go to latest partition	⍰ time-bounded scans can be pruned	⍰ new tablets can be added for future time periods
`hash(host, metric)`	⍰ writes are spread evenly among tablets	⍰ scans on specific hosts and metrics can be pruned	⍰ tablets could grow too large

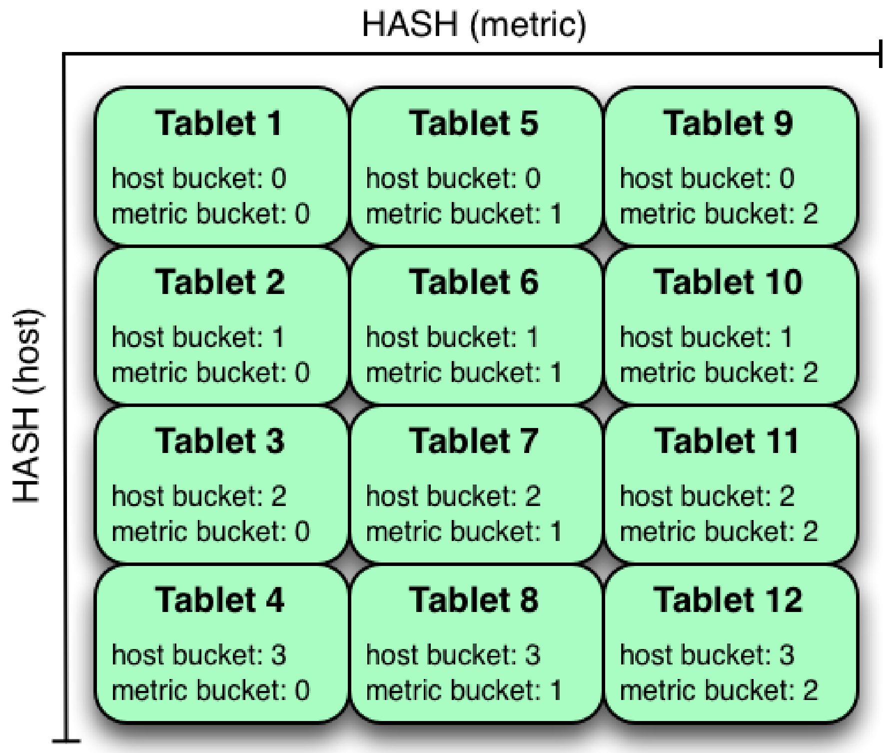
Hash partitioning is good at maximizing write throughput, while range partitioning avoids issues of unbounded tablet growth. Both strategies can take advantage of partition pruning to optimize scans in different scenarios. Using multilevel partitioning, it is possible to combine the two strategies in order to gain the benefits of both, while minimizing the drawbacks of each.



In the example above, range partitioning on the `time` column is combined with hash partitioning on the `host` and `metric` columns. This strategy can be thought of as having two dimensions of partitioning: one for the hash level and one for the range level. Writes into this table at the current time will be parallelized up to the number of hash buckets, in this case 4. Reads can take advantage of time bound **and** specific host and metric predicates to prune partitions. New range partitions can be added, which results in creating 4 additional tablets (as if a new column were added to the diagram).

Hash and Hash Partitioning

Kudu can support any number of hash partitioning levels in the same table, as long as the levels have no hashed columns in common.



In the example above, the table is hash partitioned on `host` into 4 buckets, and hash partitioned on `metric` into 3 buckets, resulting in 12 tablets. Although writes will tend to be spread among all tablets when using this strategy, it is slightly more prone to hot-spotting than when hash partitioning over multiple independent columns, since all values for an individual host or metric will always belong to a single tablet. Scans can take advantage of equality predicates on the `host` and `metric` columns separately to prune partitions.

Multiple levels of hash partitioning can also be combined with range partitioning, which logically adds another dimension of partitioning.

Schema Alterations

You can alter a table's schema in the following ways:

- Rename the table
- Rename, add, or drop non-primary key columns
- Add and drop range partitions

Multiple alteration steps can be combined in a single transactional operation.

Schema Design Limitations

Kudu currently has some known limitations that may factor into schema design. For a complete list, see Apache Kudu Schema Design and Usage Limitations on page 32.

Apache Kudu Transaction Semantics

This is a brief introduction to Kudu's transaction and consistency semantics. Kudu's core philosophy is to provide transactions with simple, strong semantics, without sacrificing performance or the ability to tune to different requirements. Kudu's transactional semantics and architecture are inspired by state-of-the-art systems such as Spanner and Calvin. For an in-depth technical exposition of what is mentioned here, see the technical report.

Kudu currently allows the following operations:

- **Scans** are read operations that can traverse multiple tablets and read information with some consistency or correctness guarantees. Scans can also perform time-travel reads. That is, you can set a scan timestamp from the past and get back results that reflect the state of the storage engine at that point in time.

  Write operations are sets of rows to be inserted, updated, or deleted in the storage engine, in a single tablet with multiple replicas. Write operations do not have separate "read sets", that is, they do not scan existing data before performing the write. Each write is only concerned with the previous state of the rows that are about to change. Writes are not "committed" explicitly by the user. Instead, they are committed automatically by the system, after completion.

While Kudu is designed to eventually be fully ACID (*Atomic, Consistent, Isolated, Durable*), multi-tablet transactions have not yet been implemented. As such, the following discussion focuses on single-tablet write operations, and only briefly touches multi-tablet reads.

Single Tablet Write Operations

Kudu employs Multiversion Concurrency Control (MVCC) and the Raft consensus algorithm. Each write operation in Kudu must go through the following order of operations:

1. The tablet's leader acquires all locks for the rows that it will change.
2. The leader assigns the write a timestamp before the write is submitted for replication. This timestamp will be the write's *tag* in MVCC.
3. After a majority of replicas have acknowledged the write, the rows are changed.
4. After the changes are complete, they are made visible to concurrent writes and reads, atomically.

All replicas of a tablet observe the same process. Therefore, if a write operation is assigned timestamp n, and changes row x, a second write operation at timestamp $m > n$ is guaranteed to see the new value of x.

This strict ordering of lock acquisition and timestamp assignment is enforced to be consistent across all replicas of a tablet through consensus. Therefore, write operations are ordered with regard to clock-assigned timestamps, relative to other writes in the same tablet. In other words, writes have strict-serializable semantics.

In case of multi-row write operations, while they are Isolated and Durable in an ACID sense, they are not yet fully Atomic. The failure of a single write in a batch operation will not roll back the entire operation, but produce per-row errors.

Writing to Multiple Tablets

Kudu does not support transactions that span multiple tablets. However, consistent snapshot reads are possible (with caveats, as explained below). Writes from a Kudu client are optionally buffered in memory until they are flushed and sent to the tablet server. When a client's session is flushed, the rows for each tablet are batched together, and sent to the tablet server which hosts the leader replica of the tablet. Since there are no inter-tablet transactions, each of these batches represents a single, independent write operation with its own timestamp. However, the client API provides the option to impose some constraints on the assigned timestamps and on how writes to different tablets are observed by clients.

Kudu was designed to be externally consistent, that is, preserving consistency even when operations span multiple tablets and even multiple data centers. In practice this means that if a write operation changes item *x* at tablet *A*, and a following write operation changes item *y* at tablet *B*, you might want to enforce that if the change to *y* is observed, the change to *x* must also be observed. There are many examples where this can be important. For example, if Kudu is storing clickstreams for further analysis, and two clicks follow each other but are stored in different tablets, subsequent clicks should be assigned subsequent timestamps so that the causal relationship between them is captured.

- **`CLIENT_PROPAGATED` Consistency**

  Kudu's default external consistency mode is called `CLIENT_PROPAGATED`. This mode causes writes from *a single client* to be automatically externally consistent. In the clickstream scenario above, if the two clicks are submitted by different client instances, the application must manually propagate timestamps from one client to the other for the causal relationship to be captured. Timestamps between clients *a* and *b* can be propagated as follows:

  Java Client

  Call `AsyncKuduClient#getLastPropagatedTimestamp()` on client *a*, propagate the timestamp to client *b*, and call `AsyncKuduClient#setLastPropagatedTimestamp()` on client *b*.

  C++ Client

  Call `KuduClient::GetLatestObservedTimestamp()` on client *a*, propagate the timestamp to client *b*, and call `KuduClient::SetLatestObservedTimestamp()` on client *b*.

- **`COMMIT_WAIT` Consistency**

  Kudu also has an experimental implementation of an external consistency model (used in Google's Spanner), called `COMMIT_WAIT`. `COMMIT_WAIT` works by tightly synchronizing the clocks on all machines in the cluster. Then, when a write occurs, timestamps are assigned and the results of the write are not made visible until enough time has passed so that no other machine in the cluster could possibly assign a lower timestamp to a following write.

  When using this mode, the latency of writes is tightly tied to the accuracy of clocks on all the cluster hosts, and using this mode with loose clock synchronization causes writes to either take a long time to complete, or even time out.

  The `COMMIT_WAIT` consistency mode may be selected as follows:

  Java Client

  Call `KuduSession#setExternalConsistencyMode(ExternalConsistencyMode.COMMIT_WAIT)`

  C++ Client

  Call `KuduSession::SetExternalConsistencyMode(COMMIT_WAIT)`

  > **Warning:**
  >
  > `COMMIT_WAIT` consistency is an experimental feature. It may return incorrect results, exhibit performance issues, or negatively impact cluster stability. Its use in production environments is discouraged.

Read Operations (Scans)

Scans are read operations performed by clients that may span one or more rows across one or more tablets. When a server receives a scan request, it takes a snapshot of the MVCC state and then proceeds in one of two ways depending on the read mode selected by the user. The mode may be selected as follows:

Java Client

Call `KuduScannerBuilder#setReadMode(…)`

C++ Client

Call `KuduScanner::SetReadMode()`

The following modes are available in both clients:

`READ_LATEST`

This is the default read mode. The server takes a snapshot of the MVCC state and proceeds with the read immediately. Reads in this mode only yield 'Read Committed' isolation.

`READ_AT_SNAPSHOT`

In this read mode, scans are consistent and repeatable. A timestamp for the snapshot is selected either by the server, or set explicitly by the user through `KuduScanner::SetSnapshotMicros()`. Explicitly setting the timestamp is recommended.

The server waits until this timestamp is 'safe'; that is, until all write operations that have a lower timestamp have completed and are visible). This delay, coupled with an external consistency method, will eventually allow Kudu to have full `strict-serializable` semantics for reads and writes. However, this is still a work in progress and some anomalies are still possible. Only scans in this mode can be fault-tolerant.

Selecting between read modes requires balancing the trade-offs and making a choice that fits your workload. For instance, a reporting application that needs to scan the entire database might need to perform careful accounting operations, so that scan may need to be fault-tolerant, but probably doesn't require a to-the-microsecond up-to-date view of the database. In that case, you might choose READ_AT_SNAPSHOT and select a timestamp that is a few seconds in the past when the scan starts. On the other hand, a machine learning workload that is not ingesting the whole data set and is already statistical in nature might not require the scan to be repeatable, so you might choose READ_LATEST instead.

Known Issues and Limitations

There are several gaps and corner cases that currently prevent Kudu from being strictly-serializable in certain situations.

Reads (Scans)

Support for `COMMIT_WAIT` is experimental and requires careful tuning of the time-synchronization protocol, such as NTP (Network Time Protocol). Its use in production environments is discouraged.

Recommendation

If external consistency is a requirement and you decide to use `COMMIT_WAIT`, the time-synchronization protocol needs to be tuned carefully. Each transaction will wait 2x the maximum clock error at the time of execution, which is usually in the 100 msec. to 1 sec. range with the default settings, maybe more. Thus, transactions would take at least 200 msec. to 2 sec. to complete when using the default settings and may even time out.

- A local server should be used as a time server. We've performed experiments using the default NTP time source available in a Google Compute Engine data center and were able to obtain a reasonable tight max error bound, usually varying between 12-17 milliseconds.
- The following parameters should be adjusted in `/etc/ntp.conf` to tighten the maximum error:
  - `server my_server.org iburst minpoll 1 maxpoll 8`
  - `tinker dispersion 500`
  - `tinker allan 0`

Writes

- On a leader change, READ_AT_SNAPSHOT scans at a snapshot whose timestamp is beyond the last write, may yield non-repeatable reads (see KUDU-1188).

Recommendation

If repeatable snapshot reads are a requirement, use READ_AT_SNAPSHOT with a timestamp that is slightly in the past (between 2-5 seconds, ideally). This will circumvent the anomaly described above. Even when the anomaly has been addressed, back-dating the timestamp will always make scans faster, since they are unlikely to block.

- Impala scans are currently performed as READ_LATEST and have no consistency guarantees.
- In AUTO_BACKGROUND_FLUSH mode, or when using "async" flushing mechanisms, writes applied to a single client session may get reordered due to the concurrency of flushing the data to the server. This is particularly noticeable if a single row is quickly updated with different values in succession. This phenomenon affects all client API implementations. Workarounds are described in the respective API documentation for FlushMode or AsyncKuduSession. See KUDU-1767.

Troubleshooting Apache Kudu

This guide covers basic Apache Kudu troubleshooting information. For more details, see the [official Kudu documentation for troubleshooting]().

Issues Starting or Restarting the Master or Tablet Server

Error during hole punch test

Kudu requires hole punching capabilities in order to be efficient. Support for hole punching depends on your operating system kernel version and local filesystem. On Linux, hole punching is the use of the `fallocate()` system call with the `FALLOC_FL_PUNCH_HOLE` option set.

- RHEL or CentOS 6.4 or later, patched to kernel version of 2.6.32-358 or later. Unpatched RHEL or CentOS 6.4 does not include a kernel with support for hole punching.
- Ubuntu 14.04 includes version 3.13 of the Linux kernel, which supports hole punching.
- Newer versions of the EXT4 or XFS filesystems support hole punching, but EXT3 does not. Older versions of XFS that do not support hole punching return a `EOPNOTSUPP` (operation not supported) error. Older versions of either EXT4 or XFS that do not support hole punching cause Kudu to emit an error message such as the following and to fail to start:

```
Error during hole punch test. The log block manager requires a
filesystem with hole punching support such as ext4 or xfs. On el6,
kernel version 2.6.32-358 or newer is required. To run without hole
punching (at the cost of some efficiency and scalability), reconfigure
Kudu with --block_manager=file. Refer to the Kudu documentation for more
details. Raw error message follows.
```

- Without hole punching support, the log block manager will never delete blocks and progressively occupy even more space on disk, which makes it unsafe to use.
- If you can't use hole punching in your environment, you can still try Kudu. Enable the file block manager instead of the log block manager by adding the `--block_manager=file` flag to the commands you use to start the master and tablet servers. Note that the file block manager does not scale as well as the log block manager, and should only be used for small-scale deployments.

Clock is not synchronized

The clock on each Kudu master and tablet server daemon must be synchronized using Network Time Protocol (NTP). If NTP is not installed or is not running, you may see errors such as the following:

```
I0929 10:00:26.570979 21371 master_main.cc:52] Initializing master server...
F0929 10:00:26.571107 21371 master_main.cc:53] Check failed: _s.ok() Bad status: Service
 unavailable: Clock is not synchronized:
  Error reading clock. Clock considered unsynchronized. Errno: Invalid argument
```

```
let_server_main.cc:48] Initializing tablet server...
F0929 10:00:26.572041 21370 tablet_server_main.cc:49] Check failed: _s.ok() Bad status:
 Service unavailable: Clock is not synchronized:
  Error reading clock. Clock considered unsynchronized. Errno: Success
```

To resolve such errors, make sure that NTP is installed on each master and tablet server, and that all NTP processes synchronize to the same time source.

- To install NTP, use the command appropriate for your operating system:

OS	Command
Debian/Ubuntu	`sudo apt-get install ntp`
RHEL/CentOS	`sudo yum install ntp`

- If NTP is installed but the clock is reported as unsynchronized, Kudu will not start, and will emit a message such as:

```
F0924 20:24:36.336809 14550 hybrid_clock.cc:191 Couldn't get the current time: Clock
unsynchronized. Status: Service unavailable: Error reading clock. Clock considered
unsynchronized.
```

  You can monitor clock synchronization status by running the `ntptime` command. The relevant value is what is reported for `maximum error`. Note that NTP requires a network connection and may take a few minutes to synchronize the clock. In some cases a spotty network connection may make NTP report the clock as unsynchronized. A common, though temporary, workaround for this is to restart NTP with one of the following commands.

OS	Command
Debian/Ubuntu	`sudo service ntp restart`
RHEL/CentOS	`sudo /etc/init.d/ntpd restart`

- In addition to the clocks being synchronized, the **`maximum clock error`** (not to be mistaken with the estimated error) must be set to a value relevant to your deployment. The default value is 10 seconds, but it can be configured using the `--max_clock_sync_error_usec` flag.

  If NTP is installed and synchronized, but the maximum clock error is too high, you will see a message such as:

```
Sep 17, 8:13:09.873 PM FATAL hybrid_clock.cc:196 Couldn't get the current time: Clock
synchronized, but error: 11130000, is past the maximum allowable error: 10000000
```

  or

```
Sep 17, 8:32:31.135 PM FATAL tablet_server_main.cc:38 Check failed: _s.ok() Bad status:
 Service unavailable: Cannot initialize clock: Cannot initialize HybridClock. Clock
synchronized but error was too high (11711000 us).
```

  If NTP reports the clock as synchronized, but the maximum error is consistently too high, you can increase the threshold to a higher value by setting the `max_clock_sync_error_usec` flag. For example, to increase the maximum error to 20 seconds, set the flag as follows: `--max_clock_sync_error_usec=20000000`.

deploy.py script exits with the *too few arguments* error

The `deploy.py` script, which is used to create a new Impala_Kudu service or clone one from an existing Apache Impala (incubating) service, takes a different number of required arguments when using the `create` and `clone` operations.

If you use the `create` option without specifying scratch directories, the `deploy.py` script will fail with the error `create: error: too few arguments`. Re-run the command, specifying scratch directories:

```
deploy.py create <scratch_dirs> <service_name> --cluster <cluster_name>

deploy.py create /data/impala IMPALA_KUDU-1 --cluster 'Cluster 1'
```

Troubleshooting Performance Issues

Kudu Tracing

The Kudu master and tablet server daemons include built-in support for tracing based on the open source Chromium Tracing framework. You can use tracing to diagnose latency issues or other problems on Kudu servers.

Accessing the Tracing Web Interface

The tracing interface is part of the embedded web server in each of the Kudu daemons, and can be accessed using a web browser. Note that while the interface has been known to work in recent versions of Google Chrome, other browsers may not work as expected.

Daemon	URL
Tablet Server	`<tablet-server-1.example.com>:8050/tracing.html`
Master	`<master-1.example.com>:8051/tracing.html`

Saving Traces

After you have collected traces, you can save these traces as JSON files by clicking **Save**. To load and analyze a saved JSON file, click **Load** and choose the file.

RPC Timeout Traces

If client applications are experiencing RPC timeouts, the Kudu tablet server `WARNING` level logs should contain a log entry which includes an RPC-level trace. For example:

```
W0922 00:56:52.313848 10858 inbound_call.cc:193] Call
kudu.consensus.ConsensusService.UpdateConsensus
from 192.168.1.102:43499 (request call id 3555909) took 1464ms (client timeout 1000).
W0922 00:56:52.314888 10858 inbound_call.cc:197] Trace:
0922 00:56:50.849505 (+     0us) service_pool.cc:97] Inserting onto call queue
0922 00:56:50.849527 (+    22us) service_pool.cc:158] Handling call
0922 00:56:50.849574 (+    47us) raft_consensus.cc:1008] Updating replica for 2 ops
0922 00:56:50.849628 (+    54us) raft_consensus.cc:1050] Early marking committed up to
 term: 8 index: 880241
0922 00:56:50.849968 (+   340us) raft_consensus.cc:1056] Triggering prepare for 2 ops
0922 00:56:50.850119 (+   151us) log.cc:420] Serialized 1555 byte log entry
0922 00:56:50.850213 (+    94us) raft_consensus.cc:1131] Marking committed up to term:
 8 index: 880241
0922 00:56:50.850218 (+     5us) raft_consensus.cc:1148] Updating last received op as
term: 8 index: 880243
0922 00:56:50.850219 (+     1us) raft_consensus.cc:1195] Filling consensus response to
 leader.
0922 00:56:50.850221 (+     2us) raft_consensus.cc:1169] Waiting on the replicates to
finish logging
0922 00:56:52.313763 (+1463542us) raft_consensus.cc:1182] finished
0922 00:56:52.313764 (+     1us) raft_consensus.cc:1190] UpdateReplicas() finished
0922 00:56:52.313788 (+    24us) inbound_call.cc:114] Queueing success response
```

These traces can indicate which part of the request was slow. Make sure you include them when filing bug reports related to RPC latency outliers.

Kernel Stack Watchdog Traces

Each Kudu server process has a background thread called the Stack Watchdog, which monitors other threads in the server in case they are blocked for longer-than-expected periods of time. These traces can indicate operating system issues or bottle-necked storage.

When the watchdog thread identifies a case of thread blockage, it logs an entry in the `WARNING` log as follows:

```
W0921 23:51:54.306350 10912 kernel_stack_watchdog.cc:111] Thread 10937 stuck at
/data/kudu/consensus/log.cc:505 for 537ms:
Kernel stack:
[<ffffffffa00b209d>] do_get_write_access+0x29d/0x520 [jbd2]
[<ffffffffa00b2471>] jbd2_journal_get_write_access+0x31/0x50 [jbd2]
[<ffffffffa00fe6d8>] __ext4_journal_get_write_access+0x38/0x80 [ext4]
[<ffffffffa00d9b23>] ext4_reserve_inode_write+0x73/0xa0 [ext4]
[<ffffffffa00d9b9c>] ext4_mark_inode_dirty+0x4c/0x1d0 [ext4]
[<ffffffffa00d9e90>] ext4_dirty_inode+0x40/0x60 [ext4]
[<ffffffff811ac48b>] __mark_inode_dirty+0x3b/0x160
[<ffffffff8119c742>] file_update_time+0xf2/0x170
[<ffffffff8111c1e0>] __generic_file_aio_write+0x230/0x490
[<ffffffff8111c4c8>] generic_file_aio_write+0x88/0x100
[<ffffffffa00d3fb1>] ext4_file_write+0x61/0x1e0 [ext4]
[<ffffffff81180f5b>] do_sync_readv_writev+0xfb/0x140
[<ffffffff81181ee6>] do_readv_writev+0xd6/0x1f0
[<ffffffff81182046>] vfs_writev+0x46/0x60
[<ffffffff81182102>] sys_pwritev+0xa2/0xc0
[<ffffffff8100b072>] system_call_fastpath+0x16/0x1b
[<ffffffffffffffff>] 0xffffffffffffffff

User stack:
    @       0x3a1ace10c4  (unknown)
    @          0x1262103  (unknown)
    @          0x12622d4  (unknown)
    @          0x12603df  (unknown)
    @           0x8e7bfb  (unknown)
    @           0x8f478b  (unknown)
    @           0x8f55db  (unknown)
    @          0x12a7b6f  (unknown)
    @       0x3a1b007851  (unknown)
    @       0x3a1ace894d  (unknown)
    @              (nil)  (unknown)
```

These traces can be useful for diagnosing root-cause latency issues in Kudu especially when they are caused by underlying systems such as disk controllers or file systems.

Cloudera Manager Metrics for Kudu

The following topics list the metrics collected by Cloudera Manager when a Kudu service is managed by Cloudera Manager. For more information about metrics in Cloudera Manager, see [Cloudera Manager Metrics] and [Metric Aggregation].

Kudu Metrics

In addition to these base metrics, many aggregate metrics are available. If an entity type has parents defined, you can formulate all possible aggregate metrics using the formula *base_metric_across_parents*.

In addition, metrics for aggregate totals can be formed by adding the prefix `total_` to the front of the metric name.

Use the type-ahead feature in the Cloudera Manager chart browser to find the exact aggregate metric name, in case the plural form does not end in "s".

For example, the following metric names may be valid for Kudu:

- `alerts_rate_across_clusters`
- `total_alerts_rate_across_clusters`

Some metrics, such as `alerts_rate`, apply to nearly every metric context. Others only apply to a certain service or role.

Metric Name	Description	Unit	Parents	CDH Version
alerts_rate	The number of alerts.	events per second	cluster	CDH 4, CDH 5
events_critical_rate	The number of critical events.	events per second	cluster	CDH 4, CDH 5
events_important_rate	The number of important events.	events per second	cluster	CDH 4, CDH 5
events_informational_rate	The number of informational events.	events per second	cluster	CDH 4, CDH 5
health_bad_rate	Percentage of Time with Bad Health	seconds per second	cluster	CDH 4, CDH 5
health_concerning_rate	Percentage of Time with Concerning Health	seconds per second	cluster	CDH 4, CDH 5
health_disabled_rate	Percentage of Time with Disabled Health	seconds per second	cluster	CDH 4, CDH 5
health_good_rate	Percentage of Time with Good Health	seconds per second	cluster	CDH 4, CDH 5
health_unknown_rate	Percentage of Time with Unknown Health	seconds per second	cluster	CDH 4, CDH 5

Kudu Replica Metrics

In addition to these base metrics, many aggregate metrics are available. If an entity type has parents defined, you can formulate all possible aggregate metrics using the formula *base_metric_across_parents*.

In addition, metrics for aggregate totals can be formed by adding the prefix `total_` to the front of the metric name.

Use the type-ahead feature in the Cloudera Manager chart browser to find the exact aggregate metric name, in case the plural form does not end in "s".

For example, the following metric names may be valid for Kudu Replica:

- `kudu_all_transactions_inflight_across_clusters`
- `total_kudu_all_transactions_inflight_across_clusters`

Some metrics, such as `alerts_rate`, apply to nearly every metric context. Others only apply to a certain service or role.

For more information about metrics, see Cloudera Manager Metrics and Metric Aggregation.

Metric Name	Description	Unit	Parents	CDH Version
kudu_all_transactions_inflight	Number of transactions currently in-flight, including any type.	transactions	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_alter_schema_transactions_inflight	Number of alter schema transactions currently in-flight	transactions	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_bloom_lookups_per_op_rate	Tracks the number of bloom filter lookups performed by each operation. A single operation may perform several bloom filter lookups if the tablet is not fully compacted. High frequency of high values may indicate that compaction is falling behind. This is the total number of recorded samples.	samples per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_bloom_lookups_per_op_sum_rate	Tracks the number of bloom filter lookups performed by each operation. A single operation may perform several bloom filter lookups if the tablet is not fully compacted. High frequency of high values may indicate that compaction is falling behind. This is the total sum of recorded samples.	message.units.probes per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_bloom_lookups_rate	Number of times a bloom filter was consulted	message.units.probes per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_bytes_flushed_rate	Amount of data that has been flushed to disk by this tablet.	bytes per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5

Metric Name	Description	Unit	Parents	CDH Version
kudu_commit_wait_duration_rate	Time spent waiting for COMMIT_WAIT external consistency writes for this tablet. This is the total number of recorded samples.	samples per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_commit_wait_duration_sum_rate	Time spent waiting for COMMIT_WAIT external consistency writes for this tablet. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_compact_rs_duration_rate	Time spent compacting RowSets. This is the total number of recorded samples.	samples per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_compact_rs_duration_sum_rate	Time spent compacting RowSets. This is the total sum of recorded samples.	message.units.milliseconds per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_compact_rs_running	Number of RowSet compactions currently running.	operations	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_delta_file_lookups_per_op_rate	Tracks the number of delta file lookups performed by each operation. A single operation may perform several delta file lookups if the tablet is not fully compacted. High frequency of high values may indicate that compaction is falling behind. This is the total number of recorded samples.	samples per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_delta_file_lookups_per_op_sum_rate	Tracks the number of delta file lookups performed by each operation. A single operation may perform several delta file lookups if the tablet is not fully compacted. High frequency of high values may indicate that compaction is falling behind. This is the total sum of recorded samples.	message.units.probes per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_delta_file_lookups_rate	Number of times a delta file was consulted	message.units.probes per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5

Metric Name	Description	Unit	Parents	CDH Version
kudu_delta_major_compact_rs_duration_rate	Seconds spent major delta compacting. This is the total number of recorded samples.	samples per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_delta_major_compact_rs_duration_sum_rate	Seconds spent major delta compacting. This is the total sum of recorded samples.	seconds per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_delta_major_compact_rs_running	Number of delta major compactions currently running.	operations	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_delta_minor_compact_rs_duration_rate	Time spent minor delta compacting. This is the total number of recorded samples.	samples per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_delta_minor_compact_rs_duration_sum_rate	Time spent minor delta compacting. This is the total sum of recorded samples.	message.units.milliseconds per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_delta_minor_compact_rs_running	Number of delta minor compactions currently running.	operations	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_flush_dms_duration_rate	Time spent flushing DeltaMemStores. This is the total number of recorded samples.	samples per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_flush_dms_duration_sum_rate	Time spent flushing DeltaMemStores. This is the total sum of recorded samples.	message.units.milliseconds per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_flush_dms_running	Number of delta memstore flushes currently running.	operations	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_flush_mrs_duration_rate	Time spent flushing MemRowSets. This is the total number of recorded samples.	samples per second	cluster, kudu, kudu-kudu_tserver, kudu_table,	CDH 4, CDH 5

Metric Name	Description	Unit	Parents	CDH Version
			kudu_tablet, rack	
kudu_flush_mrs_duration_sum_rate	Time spent flushing MemRowSets. This is the total sum of recorded samples.	message.units.milliseconds per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_flush_mrs_running	Number of MemRowSet flushes currently running.	operations	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_follower_memory_pressure_rejections_rate	Number of RPC requests rejected due to memory pressure while FOLLOWER.	requests per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_in_progress_ops	Number of operations in the peer's queue ack'd by a minority of peers.	operations	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_insertions_failed_dup_key_rate	Number of inserts which failed because the key already existed	message.units.rows per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_key_file_lookups_per_op_rate	Tracks the number of key file lookups performed by each operation. A single operation may perform several key file lookups if the tablet is not fully compacted and if bloom filters are not effectively culling lookups. This is the total number of recorded samples.	samples per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_key_file_lookups_per_op_sum_rate	Tracks the number of key file lookups performed by each operation. A single operation may perform several key file lookups if the tablet is not fully compacted and if bloom filters are not effectively culling lookups. This is the total sum of recorded samples.	message.units.probes per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_key_file_lookups_rate	Number of times a key cfile was consulted	message.units.probes per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5

Metric Name	Description	Unit	Parents	CDH Version
kudu_leader_memory_pressure_rejections_rate	Number of RPC requests rejected due to memory pressure while LEADER.	requests per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_log_append_latency_rate	Microseconds spent on appending to the log segment file. This is the total number of recorded samples.	samples per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_log_append_latency_sum_rate	Microseconds spent on appending to the log segment file. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_log_bytes_logged_rate	Number of bytes logged since service start	bytes per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_log_cache_num_ops	Number of operations in the log cache.	operations	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_log_cache_size	Amount of memory in use for caching the local log.	bytes	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_log_entry_batches_per_group_rate	Number of log entry batches in a group commit group. This is the total number of recorded samples.	samples per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_log_entry_batches_per_group_sum_rate	Number of log entry batches in a group commit group. This is the total sum of recorded samples.	requests per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_log_gc_duration_rate	Time spent garbage collecting the logs. This is the total number of recorded samples.	samples per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_log_gc_duration_sum_rate	Time spent garbage collecting the logs. This is the total sum of recorded samples.	message.units.milliseconds per second	cluster, kudu, kudu-kudu_tserver, kudu_table,	CDH 4, CDH 5

Metric Name	Description	Unit	Parents	CDH Version
			kudu_tablet, rack	
kudu_log_gc_running	Number of log GC operations currently running.	operations	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_log_group_commit_latency_rate	Microseconds spent on committing an entire group. This is the total number of recorded samples.	samples per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_log_group_commit_latency_sum_rate	Microseconds spent on committing an entire group. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_log_reader_bytes_read_rate	Data read from the WAL since tablet start	bytes per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_log_reader_entries_read_rate	Number of entries read from the WAL since tablet start	entries per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_log_reader_read_batch_latency_rate	Microseconds spent reading log entry batches. This is the total number of recorded samples.	samples per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_log_reader_read_batch_latency_sum_rate	Microseconds spent reading log entry batches. This is the total sum of recorded samples.	bytes per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_log_roll_latency_rate	Microseconds spent on rolling over to a new log segment file. This is the total number of recorded samples.	samples per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_log_roll_latency_sum_rate	Microseconds spent on rolling over to a new log segment file. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5

Metric Name	Description	Unit	Parents	CDH Version
kudu_log_sync_latency_rate	Microseconds spent on synchronizing the log segment file. This is the total number of recorded samples.	samples per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_log_sync_latency_sum_rate	Microseconds spent on synchronizing the log segment file. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_majority_done_ops	Number of operations in the leader queue ack'd by a majority but not all peers. This metric is always zero for followers.	operations	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_memrowset_size	Size of this tablet's memrowset	bytes	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_mrs_lookups_rate	Number of times a MemRowSet was consulted.	message.units.probes per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_on_disk_size	Size of this tablet on disk.	bytes	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_op_prepare_queue_length_rate	Number of operations waiting to be prepared within this tablet. High queue lengths indicate that the server is unable to process operations as fast as they are being written to the WAL. This is the total number of recorded samples.	samples per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_op_prepare_queue_length_sum_rate	Number of operations waiting to be prepared within this tablet. High queue lengths indicate that the server is unable to process operations as fast as they are being written to the WAL. This is the total sum of recorded samples.	tasks per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_op_prepare_queue_time_rate	Time that operations spent waiting in the prepare queue before being processed. High queue times indicate that the	samples per second	cluster, kudu, kudu-kudu_tserver, kudu_table,	CDH 4, CDH 5

Metric Name	Description	Unit	Parents	CDH Version
	server is unable to process operations as fast as they are being written to the WAL. This is the total number of recorded samples.		kudu_tablet, rack	
kudu_op_prepare_queue_time_sum_rate	Time that operations spent waiting in the prepare queue before being processed. High queue times indicate that the server is unable to process operations as fast as they are being written to the WAL. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_op_prepare_run_time_rate	Time that operations spent being prepared in the tablet. High values may indicate that the server is under-provisioned or that operations are experiencing high contention with one another for locks. This is the total number of recorded samples.	samples per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_op_prepare_run_time_sum_rate	Time that operations spent being prepared in the tablet. High values may indicate that the server is under-provisioned or that operations are experiencing high contention with one another for locks. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_raft_term	Current Term of the Raft Consensus algorithm. This number increments each time a leader election is started.	message.units.units	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_rows_deleted_rate	Number of row delete operations performed on this tablet since service start	message.units.rows per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_rows_inserted_rate	Number of rows inserted into this tablet since service start	message.units.rows per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_rows_updated_rate	Number of row update operations performed on this tablet since service start	message.units.rows per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5

Metric Name	Description	Unit	Parents	CDH Version
kudu_rows_upserted_rate	Number of rows upserted into this tablet since service start	message.units.rows per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_scanner_bytes_returned_rate	Number of bytes returned by scanners to clients. This count is measured after predicates are applied and the data is decoded for consumption by clients, and thus is not a reflection of the amount of work being done by scanners.	bytes per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_scanner_bytes_scanned_from_disk_rate	Number of bytes read by scan requests. This is measured as a raw count prior to application of predicates, deleted data,or MVCC-based filtering. Thus, this is a better measure of actual IO that has been caused by scan operations compared to the Scanner Bytes Returned metric. Note that this only counts data that has been flushed to disk, and does not include data read from in-memory stores. However, itincludes both cache misses and cache hits.	bytes per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_scanner_cells_returned_rate	Number of table cells returned by scanners to clients. This count is measured after predicates are applied, and thus is not a reflection of the amount of work being done by scanners.	message.units.cells per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_scanner_cells_scanned_from_disk_rate	Number of table cells processed by scan requests. This is measured as a raw count prior to application of predicates, deleted data,or MVCC-based filtering. Thus, this is a better measure of actual table cells that have been processed by scan operations compared to the Scanner Cells Returned metric. Note that this only counts data that has been flushed to disk, and does not include data read from in-memory stores. However, itincludes both cache misses and cache hits.	message.units.cells per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5

Metric Name	Description	Unit	Parents	CDH Version
kudu_scanner_rows_returned_rate	Number of rows returned by scanners to clients. This count is measured after predicates are applied, and thus is not a reflection of the amount of work being done by scanners.	message.units.rows per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_scanner_rows_scanned_rate	Number of rows processed by scan requests. This is measured as a raw count prior to application of predicates, deleted data,or MVCC-based filtering. Thus, this is a better measure of actual table rows that have been processed by scan operations compared to the Scanner Rows Returned metric.	message.units.rows per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_scans_started_rate	Number of scanners which have been started on this tablet	message.units.scanners per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_snapshot_read_inflight_wait_duration_rate	Time spent waiting for in-flight writes to complete for READ_AT_SNAPSHOT scans. This is the total number of recorded samples.	samples per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_snapshot_read_inflight_wait_duration_sum_rate	Time spent waiting for in-flight writes to complete for READ_AT_SNAPSHOT scans. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_transaction_memory_pressure_rejections_rate	Number of transactions rejected because the tablet's transaction memory limit was reached.	transactions per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_write_op_duration_client_propagated_consistency_rate	Duration of writes to this tablet with external consistency set to CLIENT_PROPAGATED. This is the total number of recorded samples.	samples per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_write_op_duration_client_propagated_consistency_sum_rate	Duration of writes to this tablet with external consistency set to CLIENT_PROPAGATED. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_write_op_duration_commit_wait_consistency_rate	Duration of writes to this tablet with external consistency set to COMMIT_WAIT. This is the total number of recorded samples.	samples per second	cluster, kudu, kudu-kudu_tserver, kudu_table,	CDH 4, CDH 5

Metric Name	Description	Unit	Parents	CDH Version
			kudu_tablet, rack	
kudu_write_op_duration_commit_wait_consistency_sum_rate	Duration of writes to this tablet with external consistency set to COMMIT_WAIT. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5
kudu_write_transactions_inflight	Number of write transactions currently in-flight	transactions	cluster, kudu, kudu-kudu_tserver, kudu_table, kudu_tablet, rack	CDH 4, CDH 5

Tablet Server Metrics

In addition to these base metrics, many aggregate metrics are available. If an entity type has parents defined, you can formulate all possible aggregate metrics using the formula *base_metric*_across_*parents*.

In addition, metrics for aggregate totals can be formed by adding the prefix `total_` to the front of the metric name.

Use the type-ahead feature in the Cloudera Manager chart browser to find the exact aggregate metric name, in case the plural form does not end in "s".

For example, the following metric names may be valid for Tablet Server:

- `alerts_rate_across_accumulos`
- `total_alerts_rate_across_accumulos`

Some metrics, such as `alerts_rate`, apply to nearly every metric context. Others only apply to a certain service or role.

Metric Name	Description	Unit	Parents	CDH Version
alerts_rate	The number of alerts.	events per second	accumulo, cluster, rack	CDH 4, CDH 5
cgroup_cpu_system_rate	CPU usage of the role's cgroup	seconds per second	accumulo, cluster, rack	CDH 4, CDH 5
cgroup_cpu_user_rate	User Space CPU usage of the role's cgroup	seconds per second	accumulo, cluster, rack	CDH 4, CDH 5
cgroup_mem_page_cache	Page cache usage of the role's cgroup	bytes	accumulo, cluster, rack	CDH 4, CDH 5
cgroup_mem_rss	Resident memory of the role's cgroup	bytes	accumulo, cluster, rack	CDH 4, CDH 5
cgroup_mem_swap	Swap usage of the role's cgroup	bytes	accumulo, cluster, rack	CDH 4, CDH 5
cgroup_read_bytes_rate	Bytes read from all disks by the role's cgroup	bytes per second	accumulo, cluster, rack	CDH 4, CDH 5
cgroup_read_ios_rate	Number of read I/O operations from all disks by the role's cgroup	ios per second	accumulo, cluster, rack	CDH 4, CDH 5

Metric Name	Description	Unit	Parents	CDH Version
cgroup_write_bytes_rate	Bytes written to all disks by the role's cgroup	bytes per second	accumulo, cluster, rack	CDH 4, CDH 5
cgroup_write_ios_rate	Number of write I/O operations to all disks by the role's cgroup	ios per second	accumulo, cluster, rack	CDH 4, CDH 5
cpu_system_rate	Total System CPU	seconds per second	accumulo, cluster, rack	CDH 4, CDH 5
cpu_user_rate	Total CPU user time	seconds per second	accumulo, cluster, rack	CDH 4, CDH 5
events_critical_rate	The number of critical events.	events per second	accumulo, cluster, rack	CDH 4, CDH 5
events_important_rate	The number of important events.	events per second	accumulo, cluster, rack	CDH 4, CDH 5
events_informational_rate	The number of informational events.	events per second	accumulo, cluster, rack	CDH 4, CDH 5
fd_max	Maximum number of file descriptors	file descriptors	accumulo, cluster, rack	CDH 4, CDH 5
fd_open	Open file descriptors.	file descriptors	accumulo, cluster, rack	CDH 4, CDH 5
health_bad_rate	Percentage of Time with Bad Health	seconds per second	accumulo, cluster, rack	CDH 4, CDH 5
health_concerning_rate	Percentage of Time with Concerning Health	seconds per second	accumulo, cluster, rack	CDH 4, CDH 5
health_disabled_rate	Percentage of Time with Disabled Health	seconds per second	accumulo, cluster, rack	CDH 4, CDH 5
health_good_rate	Percentage of Time with Good Health	seconds per second	accumulo, cluster, rack	CDH 4, CDH 5
health_unknown_rate	Percentage of Time with Unknown Health	seconds per second	accumulo, cluster, rack	CDH 4, CDH 5
kudu_active_scanners	Number of scanners that are currently active	message.units.scanners	cluster, kudu, rack	CDH 4, CDH 5
kudu_block_cache_evictions_rate	Number of blocks evicted from the cache	blocks per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_block_cache_hits_caching_rate	Number of lookups that were expecting a block that found one.Use this number instead of cache_hits when trying to determine how efficient the cache is	blocks per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_block_cache_hits_rate	Number of lookups that found a block	blocks per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_block_cache_inserts_rate	Number of blocks inserted in the cache	blocks per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_block_cache_lookups_rate	Number of blocks looked up from the cache	blocks per second	cluster, kudu, rack	CDH 4, CDH 5

Metric Name	Description	Unit	Parents	CDH Version
kudu_block_cache_misses_caching_rate	Number of lookups that were expecting a block that didn't yield one.Use this number instead of cache_misses when trying to determine how efficient the cache is	blocks per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_block_cache_misses_rate	Number of lookups that didn't yield a block	blocks per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_block_cache_usage	Memory consumed by the block cache	bytes	cluster, kudu, rack	CDH 4, CDH 5
kudu_block_manager_blocks_open_reading	Number of data blocks currently open for reading	blocks	cluster, kudu, rack	CDH 4, CDH 5
kudu_block_manager_blocks_open_writing	Number of data blocks currently open for writing	blocks	cluster, kudu, rack	CDH 4, CDH 5
kudu_block_manager_total_bytes_read_rate	Number of bytes of block data read since service start	bytes per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_block_manager_total_bytes_written_rate	Number of bytes of block data written since service start	bytes per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_block_manager_total_readable_blocks_rate	Number of data blocks opened for reading since service start	blocks per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_block_manager_total_writable_blocks_rate	Number of data blocks opened for writing since service start	blocks per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_code_cache_hits_rate	Number of codegen cache hits since start	hits per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_code_cache_queries_rate	Number of codegen cache queries (hits + misses) since start	queries per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_cpu_stime_rate	Total system CPU time of the process	message.units.milliseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_cpu_utime_rate	Total user CPU time of the process	message.units.milliseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_generic_current_allocated_bytes	Number of bytes used by the application. This will not typically match the memory use reported by the OS, because it does not include TCMalloc overhead or memory fragmentation.	bytes	cluster, kudu, rack	CDH 4, CDH 5
kudu_generic_heap_size	Bytes of system memory reserved by TCMalloc.	bytes	cluster, kudu, rack	CDH 4, CDH 5
kudu_glog_error_messages_rate	Number of ERROR-level log messages emitted by the application.	messages per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_glog_info_messages_rate	Number of INFO-level log messages emitted by the application.	messages per second	cluster, kudu, rack	CDH 4, CDH 5

Metric Name	Description	Unit	Parents	CDH Version
kudu_glog_warning_messages_rate	Number of WARNING-level log messages emitted by the application.	messages per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_consensus_consensusservice_changeconfig_rate	Microseconds spent handling kudu.consensus.ConsensusService.ChangeConfig() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_consensus_consensusservice_changeconfig_sum_rate	Microseconds spent handling kudu.consensus.ConsensusService.ChangeConfig() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_consensus_consensusservice_getconsensusstate_rate	Microseconds spent handling kudu.consensus.ConsensusService.GetConsensusState() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_consensus_consensusservice_getconsensusstate_sum_rate	Microseconds spent handling kudu.consensus.ConsensusService.GetConsensusState() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_consensus_consensusservice_getlastopid_rate	Microseconds spent handling kudu.consensus.ConsensusService.GetLastOpId() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_consensus_consensusservice_getlastopid_sum_rate	Microseconds spent handling kudu.consensus.ConsensusService.GetLastOpId() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_consensus_consensusservice_getnodeinstance_rate	Microseconds spent handling kudu.consensus.ConsensusService.GetNodeInstance() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_consensus_consensusservice_getnodeinstance_sum_rate	Microseconds spent handling kudu.consensus.ConsensusService.GetNodeInstance() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_consensus_consensusservice_leaderstepdown_rate	Microseconds spent handling kudu.consensus.ConsensusService.LeaderStepDown() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_consensus_consensusservice_leaderstepdown_sum_rate	Microseconds spent handling kudu.consensus.ConsensusService.LeaderStepDown() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_consensus_consensusservice_requestconsensusvote_rate	Microseconds spent handling kudu.consensus.ConsensusService.RequestConsensusVote() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5

Metric Name	Description	Unit	Parents	CDH Version
kudu_handler_latency_kudu_consensus_consensusservice_requestconsensusvote_sum_rate	Microseconds spent handling kudu.consensus.ConsensusService.RequestConsensusVote() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_consensus_consensusservice_runleaderelection_rate	Microseconds spent handling kudu.consensus.ConsensusService.RunLeaderElection() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_consensus_consensusservice_runleaderelection_sum_rate	Microseconds spent handling kudu.consensus.ConsensusService.RunLeaderElection() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_consensus_consensusservice_startremotebootstrap_rate	Microseconds spent handling kudu.consensus.ConsensusService.StartRemoteBootstrap() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_consensus_consensusservice_startremotebootstrap_sum_rate	Microseconds spent handling kudu.consensus.ConsensusService.StartRemoteBootstrap() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_consensus_consensusservice_updateconsensus_rate	Microseconds spent handling kudu.consensus.ConsensusService.UpdateConsensus() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_consensus_consensusservice_updateconsensus_sum_rate	Microseconds spent handling kudu.consensus.ConsensusService.UpdateConsensus() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_master_masterservice_altertable_rate	Microseconds spent handling kudu.master.MasterService.AlterTable() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_master_masterservice_altertable_sum_rate	Microseconds spent handling kudu.master.MasterService.AlterTable() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_master_masterservice_createtable_rate	Microseconds spent handling kudu.master.MasterService.CreateTable() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_master_masterservice_createtable_sum_rate	Microseconds spent handling kudu.master.MasterService.CreateTable() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_master_masterservice_deletetable_rate	Microseconds spent handling kudu.master.MasterService.DeleteTable() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5

Metric Name	Description	Unit	Parents	CDH Version
kudu_handler_latency_kudu_master_masterservice_deletetable_sum_rate	Microseconds spent handling kudu.master.MasterService.DeleteTable() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_master_masterservice_getmasterregistration_rate	Microseconds spent handling kudu.master.MasterService.GetMasterRegistration() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_master_masterservice_getmasterregistration_sum_rate	Microseconds spent handling kudu.master.MasterService.GetMasterRegistration() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_master_masterservice_gettablelocations_rate	Microseconds spent handling kudu.master.MasterService.GetTableLocations() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_master_masterservice_gettablelocations_sum_rate	Microseconds spent handling kudu.master.MasterService.GetTableLocations() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_master_masterservice_gettableschema_rate	Microseconds spent handling kudu.master.MasterService.GetTableSchema() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_master_masterservice_gettableschema_sum_rate	Microseconds spent handling kudu.master.MasterService.GetTableSchema() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_master_masterservice_gettabletlocations_rate	Microseconds spent handling kudu.master.MasterService.GetTabletLocations() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_master_masterservice_gettabletlocations_sum_rate	Microseconds spent handling kudu.master.MasterService.GetTabletLocations() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_master_masterservice_isaltertabledone_rate	Microseconds spent handling kudu.master.MasterService.IsAlterTableDone() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_master_masterservice_isaltertabledone_sum_rate	Microseconds spent handling kudu.master.MasterService.IsAlterTableDone() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_master_masterservice_iscreatetabledone_rate	Microseconds spent handling kudu.master.MasterService.IsCreateTableDone() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5

Metric Name	Description	Unit	Parents	CDH Version
kudu_handler_latency_kudu_master_masterservice_iscreatetabledone_sum_rate	Microseconds spent handling kudu.master.MasterService.IsCreateTableDone() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_master_masterservice_listmasters_rate	Microseconds spent handling kudu.master.MasterService.ListMasters() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_master_masterservice_listmasters_sum_rate	Microseconds spent handling kudu.master.MasterService.ListMasters() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_master_masterservice_listtables_rate	Microseconds spent handling kudu.master.MasterService.ListTables() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_master_masterservice_listtables_sum_rate	Microseconds spent handling kudu.master.MasterService.ListTables() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_master_masterservice_listtabletservers_rate	Microseconds spent handling kudu.master.MasterService.ListTabletServers() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_master_masterservice_listtabletservers_sum_rate	Microseconds spent handling kudu.master.MasterService.ListTabletServers() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_master_masterservice_ping_rate	Microseconds spent handling kudu.master.MasterService.Ping() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_master_masterservice_ping_sum_rate	Microseconds spent handling kudu.master.MasterService.Ping() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_master_masterservice_tsheartbeat_rate	Microseconds spent handling kudu.master.MasterService.TSHeartbeat() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_master_masterservice_tsheartbeat_sum_rate	Microseconds spent handling kudu.master.MasterService.TSHeartbeat() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_server_genericservice_flushcoverage_rate	Microseconds spent handling kudu.server.GenericService.FlushCoverage() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5

Metric Name	Description	Unit	Parents	CDH Version
kudu_handler_latency_kudu_server_genericservice_flushcoverage_sum_rate	Microseconds spent handling kudu.server.GenericService.FlushCoverage() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_server_genericservice_getstatus_rate	Microseconds spent handling kudu.server.GenericService.GetStatus() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_server_genericservice_getstatus_sum_rate	Microseconds spent handling kudu.server.GenericService.GetStatus() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_server_genericservice_serverclock_rate	Microseconds spent handling kudu.server.GenericService.ServerClock() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_server_genericservice_serverclock_sum_rate	Microseconds spent handling kudu.server.GenericService.ServerClock() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_server_genericservice_setflag_rate	Microseconds spent handling kudu.server.GenericService.SetFlag() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_server_genericservice_setflag_sum_rate	Microseconds spent handling kudu.server.GenericService.SetFlag() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_server_genericservice_setserverwallclockfortests_rate	Microseconds spent handling kudu.server.GenericService.SetServerWallClockForTests() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_server_genericservice_setserverwallclockfortests_sum_rate	Microseconds spent handling kudu.server.GenericService.SetServerWallClockForTests() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_tserver_remotebootstrapservice_beginremotebootstrapsession_rate	Microseconds spent handling kudu.tserver.RemoteBootstrapService.BeginRemoteBootstrapSession() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_tserver_remotebootstrapservice_beginremotebootstrapsession_sum_rate	Microseconds spent handling kudu.tserver.RemoteBootstrapService.BeginRemoteBootstrapSession() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_tserver_remotebootstrapservice_checksessionactive_rate	Microseconds spent handling kudu.tserver.RemoteBootstrapService.CheckSessionActive() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5

Metric Name	Description	Unit	Parents	CDH Version
kudu_handler_latency_kudu_tserver_remotebootstrapservice_checksessionactive_sum_rate	Microseconds spent handling kudu.tserver.RemoteBootstrapService.CheckSessionActive() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_tserver_remotebootstrapservice_endremotebootstrapsession_rate	Microseconds spent handling kudu.tserver.RemoteBootstrapService.EndRemoteBootstrapSession() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_tserver_remotebootstrapservice_endremotebootstrapsession_sum_rate	Microseconds spent handling kudu.tserver.RemoteBootstrapService.EndRemoteBootstrapSession() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_tserver_remotebootstrapservice_fetchdata_rate	Microseconds spent handling kudu.tserver.RemoteBootstrapService.FetchData() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_tserver_remotebootstrapservice_fetchdata_sum_rate	Microseconds spent handling kudu.tserver.RemoteBootstrapService.FetchData() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_tserver_tabletserveradminservice_alterschema_rate	Microseconds spent handling kudu.tserver.TabletServerAdminService.AlterSchema() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_tserver_tabletserveradminservice_alterschema_sum_rate	Microseconds spent handling kudu.tserver.TabletServerAdminService.AlterSchema() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_tserver_tabletserveradminservice_createtablet_rate	Microseconds spent handling kudu.tserver.TabletServerAdminService.CreateTablet() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_tserver_tabletserveradminservice_createtablet_sum_rate	Microseconds spent handling kudu.tserver.TabletServerAdminService.CreateTablet() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_tserver_tabletserveradminservice_deletetablet_rate	Microseconds spent handling kudu.tserver.TabletServerAdminService.DeleteTablet() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_tserver_tabletserveradminservice_deletetablet_sum_rate	Microseconds spent handling kudu.tserver.TabletServerAdminService.DeleteTablet() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_tserver_tabletserverservice_checksum_rate	Microseconds spent handling kudu.tserver.TabletServerService.Checksum() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5

Metric Name	Description	Unit	Parents	CDH Version
kudu_handler_latency_kudu_tserver_tabletserverservice_checksum_sum_rate	Microseconds spent handling kudu.tserver.TabletServerService.Checksum() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_tserver_tabletserverservice_listtablets_rate	Microseconds spent handling kudu.tserver.TabletServerService.ListTablets() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_tserver_tabletserverservice_listtablets_sum_rate	Microseconds spent handling kudu.tserver.TabletServerService.ListTablets() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_tserver_tabletserverservice_ping_rate	Microseconds spent handling kudu.tserver.TabletServerService.Ping() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_tserver_tabletserverservice_ping_sum_rate	Microseconds spent handling kudu.tserver.TabletServerService.Ping() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_tserver_tabletserverservice_scan_rate	Microseconds spent handling kudu.tserver.TabletServerService.Scan() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_tserver_tabletserverservice_scan_sum_rate	Microseconds spent handling kudu.tserver.TabletServerService.Scan() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_tserver_tabletserverservice_scannerkeepalive_rate	Microseconds spent handling kudu.tserver.TabletServerService.ScannerKeepAlive() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_tserver_tabletserverservice_scannerkeepalive_sum_rate	Microseconds spent handling kudu.tserver.TabletServerService.ScannerKeepAlive() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_tserver_tabletserverservice_write_rate	Microseconds spent handling kudu.tserver.TabletServerService.Write() RPC requests. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_handler_latency_kudu_tserver_tabletserverservice_write_sum_rate	Microseconds spent handling kudu.tserver.TabletServerService.Write() RPC requests. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_hybrid_clock_error	Server clock maximum error.	message.units.microseconds	cluster, kudu, rack	CDH 4, CDH 5
kudu_hybrid_clock_timestamp	Hybrid clock timestamp.	message.units.microseconds	cluster, kudu, rack	CDH 4, CDH 5

Metric Name	Description	Unit	Parents	CDH Version
kudu_involuntary_context_switches_rate	Total involuntary context switches	message.units.context_switches per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_log_block_manager_blocks_under_management	Number of data blocks currently under management	blocks	cluster, kudu, rack	CDH 4, CDH 5
kudu_log_block_manager_bytes_under_management	Number of bytes of data blocks currently under management	bytes	cluster, kudu, rack	CDH 4, CDH 5
kudu_log_block_manager_containers_rate	Number of log block containers	message.units.log_block_containers per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_log_block_manager_full_containers_rate	Number of full log block containers	message.units.log_block_containers per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_logical_clock_timestamp	Logical clock timestamp.	message.units.units	cluster, kudu, rack	CDH 4, CDH 5
kudu_op_apply_queue_length_rate	Number of operations waiting to be applied to the tablet. High queue lengths indicate that the server is unable to process operations as fast as they are being written to the WAL. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_op_apply_queue_length_sum_rate	Number of operations waiting to be applied to the tablet. High queue lengths indicate that the server is unable to process operations as fast as they are being written to the WAL. This is the total sum of recorded samples.	tasks per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_op_apply_queue_time_rate	Time that operations spent waiting in the apply queue before being processed. High queue times indicate that the server is unable to process operations as fast as they are being written to the WAL. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_op_apply_queue_time_sum_rate	Time that operations spent waiting in the apply queue before being processed. High queue times indicate that the server is unable to process operations as fast as they are being written to the WAL. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_op_apply_run_time_rate	Time that operations spent being applied to the tablet. High values may indicate that the server is	samples per second	cluster, kudu, rack	CDH 4, CDH 5

Metric Name	Description	Unit	Parents	CDH Version
	under-provisioned or that operations consist of very large batches. This is the total number of recorded samples.			
kudu_op_apply_run_time_sum_rate	Time that operations spent being applied to the tablet. High values may indicate that the server is under-provisioned or that operations consist of very large batches. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_rpc_connections_accepted_rate	Number of incoming TCP connections made to the RPC server	connections per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_rpc_incoming_queue_time_rate	Number of microseconds incoming RPC requests spend in the worker queue. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_rpc_incoming_queue_time_sum_rate	Number of microseconds incoming RPC requests spend in the worker queue. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_rpcs_queue_overflow_rate	Number of RPCs dropped because the service queue was full.	requests per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_rpcs_timed_out_in_queue_rate	Number of RPCs whose timeout elapsed while waiting in the service queue, and thus were not processed.	requests per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_scanner_duration_rate	Histogram of the duration of active scanners on this tablet. This is the total number of recorded samples.	samples per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_scanner_duration_sum_rate	Histogram of the duration of active scanners on this tablet. This is the total sum of recorded samples.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_scanners_expired_rate	Number of scanners that have expired since service start	message.units.scanners per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_spinlock_contention_time_rate	Amount of time consumed by contention on internal spinlocks since the server started. If this increases rapidly, it may indicate a performance issue in Kudu internals triggered by a particular workload and warrant investigation.	message.units.microseconds per second	cluster, kudu, rack	CDH 4, CDH 5

Metric Name	Description	Unit	Parents	CDH Version
kudu_tcmalloc_current_total_thread_cache_bytes	A measure of some of the memory TCMalloc is using (for small objects).	bytes	cluster, kudu, rack	CDH 4, CDH 5
kudu_tcmalloc_max_total_thread_cache_bytes	A limit to how much memory TCMalloc dedicates for small objects. Higher numbers trade off more memory use for -- in some situations -- improved efficiency.	bytes	cluster, kudu, rack	CDH 4, CDH 5
kudu_tcmalloc_pageheap_free_bytes	Number of bytes in free, mapped pages in page heap. These bytes can be used to fulfill allocation requests. They always count towards virtual memory usage, and unless the underlying memory is swapped out by the OS, they also count towards physical memory usage.	bytes	cluster, kudu, rack	CDH 4, CDH 5
kudu_tcmalloc_pageheap_unmapped_bytes	Number of bytes in free, unmapped pages in page heap. These are bytes that have been released back to the OS, possibly by one of the MallocExtension "Release" calls. They can be used to fulfill allocation requests, but typically incur a page fault. They always count towards virtual memory usage, and depending on the OS, typically do not count towards physical memory usage.	bytes	cluster, kudu, rack	CDH 4, CDH 5
kudu_threads_running	Current number of running threads	threads	cluster, kudu, rack	CDH 4, CDH 5
kudu_threads_started_rate	Total number of threads started on this server	threads per second	cluster, kudu, rack	CDH 4, CDH 5
kudu_voluntary_context_switches_rate	Total voluntary context switches	message.units.context_switches per second	cluster, kudu, rack	CDH 4, CDH 5
mem_rss	Resident memory used	bytes	accumulo, cluster, rack	CDH 4, CDH 5
mem_swap	Amount of swap memory used by this role's process.	bytes	accumulo, cluster, rack	CDH 4, CDH 5
mem_virtual	Virtual memory used	bytes	accumulo, cluster, rack	CDH 4, CDH 5
oom_exits_rate	The number of times the role's backing process was killed due to an OutOfMemory error. This counter is only incremented if the Cloudera Manager "Kill When Out of Memory" option is enabled.	exits per second	accumulo, cluster, rack	CDH 4, CDH 5

Metric Name	Description	Unit	Parents	CDH Version
`read_bytes_rate`	The number of bytes read from the device	bytes per second	accumulo, cluster, rack	CDH 4, CDH 5
`unexpected_exits_rate`	The number of times the role's backing process exited unexpectedly.	exits per second	accumulo, cluster, rack	CDH 4, CDH 5
`uptime`	For a host, the amount of time since the host was booted. For a role, the uptime of the backing process.	seconds	accumulo, cluster, rack	CDH 4, CDH 5
`write_bytes_rate`	The number of bytes written to the device	bytes per second	accumulo, cluster, rack	CDH 4, CDH 5

More Resources for Apache Kudu

The following is a list of resources that may help you to understand some of the architectural features of Apache Kudu and columnar data storage. The links further down tend toward the academic and are not required reading in order to understand how to install, use, and administer Kudu.

Kudu Project

Read the official Kudu documentation and learn how you can get involved.

Kudu Documentation

Read the official Kudu documentation, which includes more in-depth information about installation and configuration choices.

Kudu Github Repository

Examine the Kudu source code and contribute to the project.

Kudu-Examples Github Repository

View and run several Kudu code examples, as well as the Kudu Quickstart VM.

Kudu White Paper

Read draft of the white paper discussing Kudu's architecture, written by the Kudu development team.

In Search Of An Understandable Consensus Algorithm*, Diego Ongaro and John Ousterhout, Stanford University. 2014.*

The original whitepaper describing the Raft consensus algorithm.

Column-Stores vs. Row-Stores: How Different Are They Really? *Abadi, Madden, Hachem. 2008.*

A discussion of the characteristics of column-based and row-based datastores and their characteristics under different workloads and schemas.

Support

Bug reports and feedback can be submitted through the public JIRA, our Cloudera Community Kudu forum, and a public mailing list monitored by the Kudu development team and community members. In addition, a public Slack instance is available to communicate with the team.