cloudera®

Apache Kudu Guide

Important Notice
© 2010-2021 Cloudera, Inc. All rights reserved.

Cloudera, the Cloudera logo, and any other product or
service names or slogans contained in this document are trademarks of Cloudera and its suppliers or licensors, and may not be copied, imitated or used, in whole or in part, without the prior written permission of Cloudera or the applicable trademark holder. If this documentation includes code, including but not limited to, code examples, Cloudera makes this available to you under the terms of the Apache License, Version 2.0, including any required notices. A copy of the Apache License Version 2.0, including any notices, is included herein. A copy of the Apache License Version 2.0 can also be found here: https://opensource.org/licenses/Apache-2.0

Hadoop and the Hadoop elephant logo are trademarks of the Apache Software Foundation. All other trademarks, registered trademarks, product names and company names or logos mentioned in this document are the property of their respective owners. Reference to any products, services, processes or other information, by trade name, trademark, manufacturer, supplier or otherwise does not constitute or imply endorsement, sponsorship or recommendation thereof by us.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Cloudera.

Cloudera may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Cloudera, the furnishing of this document does not give you any license to these patents, trademarks copyrights, or other intellectual property. For information about patents covering Cloudera products, see http://tiny.cloudera.com/patents.

The information in this document is subject to change without notice. Cloudera shall not be liable for any damages resulting from technical errors or omissions which may be present in this document, or from use of this document.

Cloudera, Inc.
395 Page Mill Road
Palo Alto, CA 94306
info@cloudera.com
US: 1-888-789-1488
Intl: 1-650-362-0488
www.cloudera.com

Release Information

Version: Kudu 1.5.0 / CDH 5.13.x
Date: February 4, 2021

Table of Contents

Apache Kudu Overview........8
Kudu-Impala Integration........8
Example Use Cases........9
Related Information........9

Apache Kudu Concepts and Architecture........10
Columnar Datastore........10
Raft Consensus Algorithm........10
Table........10
Tablet........10
Tablet Server........10
Master........11
Catalog Table........11
Logical Replication........11
Architectural Overview........11

Apache Kudu Requirements........13

Overview of Apache Kudu Installation and Upgrade in CDH........14
Platform Requirements........14
Installing Kudu........14
Upgrading Kudu........14

Apache Kudu Usage Limitations........15
Schema Design Limitations........15
Partitioning Limitations........16
Scaling Recommendations and Limitations........16
Server Management Limitations........16
Cluster Management Limitations........17
Replication and Backup Limitations........17
Impala Integration Limitations........17
Spark Integration Limitations........17
Security Limitations........18

Apache Kudu Configuration....................19

Configuring the Kudu Master....................19

Configuring Tablet Servers....................20

Apache Kudu Administration....................21

Starting and Stopping Kudu Processes....................21

Kudu Web Interfaces....................21

Kudu Master Web Interface....................21

Kudu Tablet Server Web Interface....................21

Common Web Interface Pages....................21

Kudu Metrics....................22

Listing Available Metrics....................22

Collecting Metrics via HTTP....................22

Collecting Metrics to a Log....................23

Common Kudu Workflows....................23

Migrating to Multiple Kudu Masters....................24

Recovering from a Dead Kudu Master in a Multi-Master Deployment....................27

Removing Kudu Masters from a Multi-Master Deployment....................30

Changing Master Hostnames....................31

Monitoring Cluster Health with ksck....................32

Bringing a Tablet That Has Lost a Majority of Replicas Back Online....................33

Rebuilding a Kudu Filesystem Layout....................34

Physical Backups of an Entire Node....................34

Scaling Storage on Kudu Master and Tablet Servers in the Cloud....................35

Managing Kudu Using Cloudera Manager....................36

Installing and Upgrading the Kudu Service....................36

Enabling Core Dump for the Kudu Service....................36

Verifying the Impala Dependency on Kudu....................36

Using the Charts Library with the Kudu Service....................36

Developing Applications With Apache Kudu....................38

Viewing the API Documentation....................38

Kudu Example Applications....................38

Maven Artifacts....................39

Building the Java Client....................39

Kudu Python Client....................39

Example Apache Impala Commands With Kudu....................40

Kudu Integration with Spark....................40

..42
Integration with MapReduce, YARN, and Other Frameworks....................................42

Using Apache Impala with Kudu..43
Impala Database Containment Model....................................43
Internal and External Impala Tables....................................43
Using Impala To Query Kudu Tables....................................44
Querying an Existing Kudu Table from Impala....................................44
Creating a New Kudu Table From Impala....................................44
Partitioning Tables....................................45
Optimizing Performance for Evaluating SQL Predicates....................................49
Inserting a Row....................................49
Updating a Row....................................50
Upserting a Row....................................50
Altering a Table....................................51
Deleting a Row....................................52
Failures During INSERT, UPDATE, UPSERT, and DELETE Operations....................................52
Altering Table Properties....................................52
Dropping a Kudu Table using Impala....................................53
Security Considerations....................................53
Known Issues and Limitations....................................53
Next Steps....................................54

Kudu Security..55
Kudu Authentication with Kerberos....................................55
Internal Private Key Infrastructure (PKI)....................................55
Authentication Tokens....................................55
Client Authentication to Secure Kudu Clusters....................................56
Scalability....................................56
Encryption....................................56
Coarse-grained Authorization....................................56
Web UI Encryption....................................57
Web UI Redaction....................................57
Log Redaction....................................57
Configuring a Secure Kudu Cluster using Cloudera Manager....................................57
Configuring a Secure Kudu Cluster using the Command Line....................................59

Apache Kudu Schema Design..60
The Perfect Schema....................................60
Column Design....................................60
Column Encoding....................................61

Column Compression......61
Primary Key Design......62
Primary Key Index......62
Considerations for Backfill Inserts......62
Partitioning......62
Range Partitioning......63
Hash Partitioning......63
Multilevel Partitioning......63
Partition Pruning......64
Partitioning Examples......64
Schema Alterations......67
Schema Design Limitations......67

Apache Kudu Transaction Semantics......68
Single Tablet Write Operations......68
Writing to Multiple Tablets......68
Read Operations (Scans)......69
Known Issues and Limitations......70
Writes......70
Reads (Scans)......71

Apache Kudu Background Maintenance Tasks......72

Troubleshooting Apache Kudu......74
Issues Starting or Restarting the Master or Tablet Server......74
Errors During Hole Punching Test......74
NTP Clock Synchronization Issues......74
Disk Space Usage......75
Reporting Kudu Crashes Using Breakpad......76
Troubleshooting Performance Issues......77
Kudu Tracing......77
Slow Name Resolution and nscd......78
Usability Issues......79
ClassNotFoundException: com.cloudera.kudu.hive.KuduStorageHandler......79
Runtime error: Could not create thread: Resource temporarily unavailable (error 11)......79
Tombstoned or STOPPED tablet replicas......79
Corruption: checksum error on CFile block......79

More Resources for Apache Kudu......80

Appendix: Apache License, Version 2.0..81

Apache Kudu Overview

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, 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 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

Kudu-Impala Integration

Apache Kudu has tight integration with Apache Impala, 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.

- **`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 designed 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 of these. 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.

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 might 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 might need to be updated or modified often as the learning takes place or as the situation being modeled changes. In addition, the scientist might 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 might 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.

Related Information

- Apache Kudu Concepts and Architecture on page 10
- Overview of Apache Kudu Installation and Upgrade in CDH on page 14
- Kudu Security on page 55
- More Resources for Apache Kudu on page 80

Apache Kudu Concepts and Architecture

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 the following 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. 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 cannot be read or written directly. Instead, it is accessible only through 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 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. They do not even need to 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 grey.

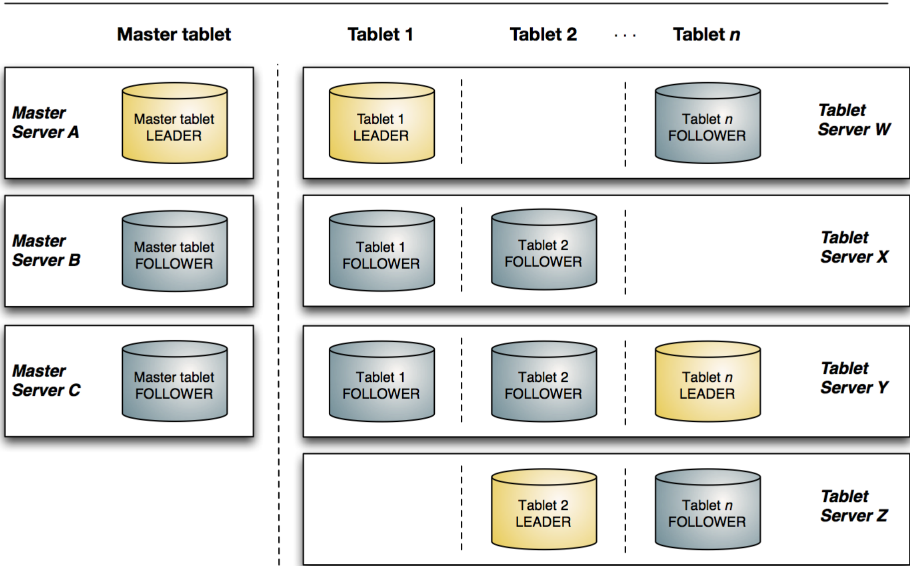


Figure 1: Kudu Architectural Overview

Apache Kudu Requirements

Starting with Kudu 1.5.0 / CDH 5.13, Kudu is fully integrated in the CDH 5 parcel and packages. As such, for the complete list of hardware and software requirements for Kudu, see the Product Compatibility Matrix for Apache Kudu.

Overview of Apache Kudu Installation and Upgrade in CDH

Starting with Apache Kudu 1.5.0 / CDH 5.13, Kudu ships with CDH 5. In a parcel-based configuration, Kudu is part of the CDH parcel rather than a separate parcel. The Kudu packages are also bundled into the CDH package.

Platform Requirements

Before you proceed with installation or upgrade:

- Review Product Compatibility Matrix - Apache Kudu.
- Review the CDH and Cloudera Manager installation options described in Cloudera Manager Deployment.

Installing Kudu

Note: Kudu is not supported in single-user mode.

On a cluster managed by Cloudera Manager, Kudu is installed as part of CDH and does not need to be installed separately. With Cloudera Manager, you can enable or disable the Kudu service, but the Kudu component remains present on the cluster. For instructions, see Installing Cloudera Manager and CDH.

On an unmanaged cluster, you can install Kudu packages manually. For instructions, see Kudu Installation.

Upgrading Kudu

Before you proceed with an upgrade, review the Upgrade Notes for Kudu 1.5.0 / CDH 5.13.0.

On a managed cluster,

- If you have just upgraded Cloudera Manager from a version that did not include Kudu, then Kudu will not be installed automatically. You will need to add the Kudu service manually. Upgrading Cloudera Manager does not automatically upgrade CDH or other managed services.
- **Parcels:** If you are upgrading CDH and were previously using the standalone Kudu parcel (version 1.4.0 and lower), then you must deactivate this parcel and activate the latest CDH parcel that includes Kudu. For instructions, see Upgrading to CDH 5.x Using Parcels.
- **Packages:** If you are upgrading CDH and were previously using the Kudu package (version 1.4.0 and lower), then you must uninstall the `kudu` package and upgrade to the latest CDH package that includes Kudu. For instructions, see Upgrading to CDH 5.x Using Packages.

On an unmanaged cluster, you can upgrade Kudu packages manually. For instructions, see Upgrade Kudu Using the Command Line.

Apache Kudu Usage Limitations

Schema Design Limitations

Primary Key

- The primary key cannot be changed after the table is created. You must drop and recreate a table to select a new primary key.
- The columns which make up the primary key must be listed first in the schema.
- The primary key of a row cannot 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 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`.
- Auto-generated primary keys are not supported.
- Cells making up a composite primary key are limited to a total of 16KB after internal composite-key encoding is done by Kudu.

Cells

No individual cell may be larger than 64KB before encoding or compression. 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.

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.
- `DECIMAL, CHAR, VARCHAR, DATE,` and complex types such as `ARRAY` are not supported.
- Type and nullability of existing columns cannot be changed by altering the table.
- Dropping a column does not immediately reclaim space. Compaction must run first.

Tables

- Tables must have an odd number of replicas, with a maximum of 7.
- Replication factor (set at table creation time) cannot be changed.
- There is no way to run compaction manually, but dropping a table will reclaim the space immediately.

Other Usage Limitations

- Secondary indexes are not supported.
- Multi-row transactions are not supported.
- Relational features, such as foreign keys, are not supported.
- Identifiers such as column and table names are restricted to be valid UTF-8 strings. Additionally, a maximum length of 256 characters is enforced.

If you are using Apache Impala to query Kudu tables, refer to the section on Impala Integration Limitations on page 17 as well.

Partitioning Limitations

- Tables must be manually pre-split into tablets using simple or compound primary keys. Automatic splitting is not yet possible. Kudu does not allow you to change how a table is partitioned after creation, with the exception of adding or dropping range partitions.
- 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.
- Tablets that lose a majority of replicas (such as 1 left out of 3) require manual intervention to be repaired.

Scaling Recommendations and Limitations

- Recommended maximum number of tablet servers is 100.
- Recommended maximum number of masters is 3.
- Recommended maximum amount of stored data, post-replication and post-compression, per tablet server is 8 TiB.
- Recommended number of tablets per tablet server is 1000 (post-replication) with 2000 being the maximum number of tablets allowed per tablet server.
- Maximum number of tablets per table for each tablet server is 60, post-replication (assuming the default replication factor of 3), at table-creation time.
- Recommended maximum amount of data per tablet is 50 GiB. Going beyond this can cause issues such a reduced performance, compaction issues, and slow tablet startup times.

  The recommended target size for tablets is under 10 GiB

Server Management Limitations

- Production deployments should configure a least 4 GiB of memory for tablet servers, and ideally more than 16 GiB when approaching the data and tablet scale limits.
- Write ahead logs (WALs) can only be stored on one disk.
- Disk failures are not tolerated and tablets servers will crash as soon as one is detected.
- Failed disks with unrecoverable data requires formatting of all Kudu data for that tablet server before it can be started again.
- Data directories cannot be added/removed; they must be reformatted to change the set of directories.
- Tablet servers cannot be gracefully decommissioned.
- Tablet servers cannot change their address or port.
- Kudu has a hard requirement on having an up-to-date NTP. Kudu masters and tablet servers will crash when out of sync.
- Kudu releases have only been tested with NTP. Other time synchronization providers such as Chrony may not work.

Cluster Management Limitations

- Rack awareness is not supported.
- Multi-datacenter is not supported.
- Rolling restart is not supported.
- All masters must be started at the same time when the cluster is started for the very first time.

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 `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 cannot be used as an external table in Impala. Columns can be renamed in Kudu to work around this issue.
- `!=` 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

- Spark 2.2 (and higher) requires Java 8 at runtime even though Kudu Spark 2.x integration is Java 7 compatible. Spark 2.2 is the default dependency version as of Kudu 1.5.0.
- 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 must not be used with SparkSQL. Columns can be renamed in Kudu to work around this issue.

- `<>` and `OR`predicates are not pushed to Kudu, and instead will be evaluated by the Spark task. Only `LIKE` predicates with a suffix wildcard are pushed to Kudu. This means `LIKE "FOO%"` will be pushed, but `LIKE "FOO%BAR"` won't.
- Kudu does not support all the types supported by Spark SQL. For example, `Date`, `Decimal`, and complex types are not supported on Kudu.
- Kudu tables can only be registered as temporary tables in SparkSQL.
- Kudu tables cannot be queried using HiveContext.

Security Limitations

- Data encryption at rest is not directly built into Kudu. Encryption of Kudu data at rest can be achieved through the use of local block device encryption software such as `dmcrypt`.
- Authorization is only available at a system-wide, coarse-grained level. Table-level, column-level, and row-level authorization features are not available.
- Kudu does not support configuring a custom service principal for Kudu processes. The principal must follow the pattern `kudu/<HOST>@<DEFAULT.REALM>`.
- Kudu integration with Apache Flume does not support writing to Kudu clusters that require authentication.

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 in 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 can 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 1: 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		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.
`--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 2: Supported Configuration Flags for Kudu Tablet Servers

Flag	Valid Options	Default	Description
--fs_data_dirs	string		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.
--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 that 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
```

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
```

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.

For the complete list of metrics collected by Cloudera Manager for a Kudu service, look for the Kudu metrics listed under Cloudera Manager Metrics .

If you are using Cloudera Manager, see Cloudera Manager Metrics for Kudu 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

The following sections describe some common workflows for Kudu users:

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. It can also be used to migrate from two masters to three with straightforward modifications.

Important:

- This workflow is unsafe for adding new masters to an existing multi-master configuration that already has three or more masters. 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. The example commands assume the Kudu Unix user is `kudu`, which is typical.

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 directories where the master's write-ahead log (WAL) and data live. If using Kudu system packages, their default locations are `/var/lib/kudu/master`, but they may be customized using the `fs_wal_dir` and `fs_data_dirs` configuration parameters. The command below assume that `fs_wal_dir` is `/data/kudu/master/wal` and `fs_data_dirs` is `/data/kudu/master/data`. Your configuration may differ. For more information on configuring these directories, see the Kudu Configuration docs.
    - 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:

```
$ sudo -u kudu kudu fs dump uuid --fs_wal_dir=<master_wal_dir>
[--fs_data_dirs=<master_data_dir>] 2>/dev/null
```

master_data_dir

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

For example:

```
$ sudo -u kudu 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 without bringing the cluster down for maintenance, and as such, it is highly recommended.

4. If you have Kudu tables that are accessed from Impala, you must update the master addresses in the Apache Hive Metastore (HMS) database.

    - If you set up the DNS aliases, run the following statement in Impala-shell, replacing `master-1`, `master-2`, and `master-3` with your actual aliases.

```
ALTER TABLE table_name
SET TBLPROPERTIES
('kudu.master_addresses' = 'master-1,master-2,master-3');
```

    - If you do not have DNS aliases set up, see Step #11 in the Performing the migration section for updating HMS.

5. 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:

```
$ sudo -u kudu kudu fs format --fs_wal_dir=<master_wal_dir>
[--fs_data_dirs=<master_data_dir>]
$ sudo -u kudu kudu fs dump uuid --fs_wal_dir=<master_wal_dir>
[--fs_data_dirs=<master_data_dir>] 2>/dev/null
```

    master_data_dir

    The new master's previously recorded data directory.

    For example:

```
$ sudo -u kudu kudu fs format --fs_wal_dir=/data/kudu/master/wal
--fs_data_dirs=/data/kudu/master/data
$ sudo -u kudu kudu fs dump uuid --fs_wal_dir=/data/kudu/master/wal
--fs_data_dirs=/data/kudu/master/data 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:

```
$ sudo -u kudu kudu local_replica cmeta rewrite_raft_config --fs_wal_dir=<master_wal_dir>
 [--fs_data_dirs=<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:

```
$ sudo -u kudu kudu local_replica cmeta rewrite_raft_config
--fs_wal_dir=/data/kudu/master/wal --fs_data_dirs=/data/kudu/master/data
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>`.

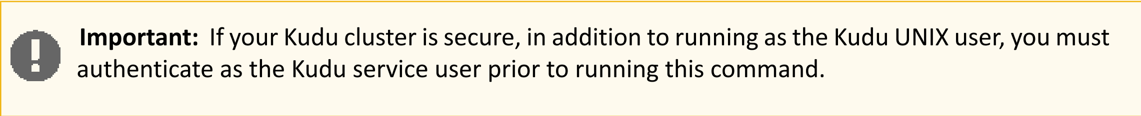
   hostname

   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.



Important: If your Kudu cluster is secure, in addition to running as the Kudu UNIX user, you must authenticate as the Kudu service user prior to running this command.

```
$ sudo -u kudu 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

```
$ sudo -u kudu kudu local_replica copy_from_remote --fs_wal_dir=/data/kudu/master/wal
--fs_data_dirs=/data/kudu/master/data  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.
11. If you have Kudu tables that are accessed from Impala and you didn’t set up DNS aliases, update the HMS database manually in the underlying database that provides the storage for HMS.

   - The following is an example SQL statement you would run in the HMS database:

```
UPDATE TABLE_PARAMS
SET PARAM_VALUE =
  'master-1.example.com,master-2.example.com,master-3.example.com'
WHERE PARAM_KEY = 'kudu.master.addresses' AND PARAM_VALUE = 'old-master';
```

   - Invalidate the metadata by running the command in Impala-shell:

```
INVALIDATE METADATA;
```

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. For more details, see Monitoring Cluster Health with ksck on page 32.

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 potentially brief one if the cluster was set up with DNS aliases.

Important:

- Kudu does not yet support live Raft configuration changes for masters. As such, it is only possible to replace a master if the deployment was created with DNS aliases or if every node in the cluster is first shut down. See the previous multi-master migration workflow for more details on deploying with DNS aliases.
- 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. If the cluster was configured without DNS aliases perform the following steps. Otherwise move on to step 2:
   a. Establish a maintenance window (one hour should be sufficient). During this time the Kudu cluster will be unavailable.
   b. Shut down all Kudu tablet server processes in the cluster.
2. 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.
3. 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.
4. Choose an unused machine in the cluster where the new master will live. The master generates very little load so it can be co-located 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.
5. 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.
6. 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. For more information on configuring these directories, see the Kudu Configuration docs.
   - Identify and record the master's UUID. It can be fetched using the following command:

```
$ sudo -u kudu kudu fs dump uuid --fs_wal_dir=<master_wal_dir>
[--fs_data_dirs=<master_data_dir>] 2>/dev/null
```

master_data_dir

live master's previously recorded data directory

Example

```
$ sudo -u kudu kudu fs dump uuid --fs_wal_dir=/data/kudu/master/wal
--fs_data_dirs=/data/kudu/master/data 2>/dev/null
80a82c4b8a9f4c819bab744927ad765c
```

7. 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. For more information on configuring these directories, see the Kudu Configuration docs.
- Identify and record the UUIDs of every master in the cluster, using the following command:

```
$ sudo -u kudu 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

```
$ sudo -u kudu kudu local_replica cmeta print_replica_uuids
--fs_wal_dir=/data/kudu/master/wal --fs_data_dirs=/data/kudu/master/data
00000000000000000000000000000000 2>/dev/null
80a82c4b8a9f4c819bab744927ad765c 2a73eeee5d47413981d9a1c637cce170
1c3f3094256347528d02ec107466aef3
```

8. 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:

```
$ sudo -u kudu kudu fs format --fs_wal_dir=<master_wal_dir>
[--fs_data_dirs=<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:

```
$ sudo -u kudu kudu fs format --fs_wal_dir=/data/kudu/master/wal
--fs_data_dirs=/data/kudu/master/data --uuid=80a82c4b8a9f4c819bab744927ad765c
```

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

Important: If your Kudu cluster is secure, in addition to running as the Kudu UNIX user, you must authenticate as the Kudu service user prior to running this command.

```
$ sudo -u kudu kudu local_replica copy_from_remote --fs_wal_dir=<master_wal_dir>
[--fs_data_dirs=<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:

```
$ sudo -u kudu kudu local_replica copy_from_remote --fs_wal_dir=/data/kudu/master/wal
--fs_data_dirs=/data/kudu/master/data 00000000000000000000000000000000 master-2:7051
```

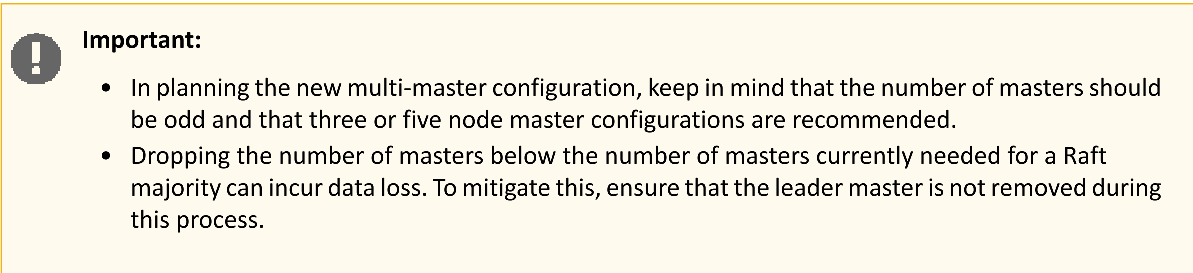
3. If you are using Cloudera Manager, add the replacement Kudu master role now, but do not start it.
   - Override the empty value of the `Master Address` parameter for the new role with the replacement master's alias.
   - If you are using a non-default RPC port, add the port number (separated by a colon) as well.
4. - If the cluster was set up *with DNS aliases*, reconfigure the DNS alias for the dead master to point at the replacement master.
   - If the cluster was set up *without DNS aliases*, perform the following steps:
     1. Stop the remaining live masters.
     2. Rewrite the Raft configurations on these masters to include the replacement master. See Step 4 of **Perform the Migration** for more details.
5. Start the replacement master.
6. Restart the remaining masters in the new multi-master deployment. While the masters are shut down, there will be an 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. For more details, see Monitoring Cluster Health with ksck on page 32.

Removing Kudu Masters from a Multi-Master Deployment

In the event that a multi-master deployment has been overallocated nodes, the following steps should be taken to remove the unwanted masters.



Important:

- In planning the new multi-master configuration, keep in mind that the number of masters should be odd and that three or five node master configurations are recommended.
- Dropping the number of masters below the number of masters currently needed for a Raft majority can incur data loss. To mitigate this, ensure that the leader master is not removed during this process.

Prepare for removal

1. Establish a maintenance window (one hour should be sufficient). During this time the Kudu cluster will be unavailable.
2. Identify the UUID and RPC address current leader of the multi-master deployment by visiting the `/masters` page of any master's web UI. This master must not be removed during this process; its removal may result in severe data loss.
3. Stop all the Kudu processes in the entire cluster.
4. If you are using Cloudera Manager, remove the unwanted Kudu master from your cluster's Kudu service.

Perform the removal

1. Rewrite the Raft configuration on the remaining masters to include only the remaining masters. See Step 4 of Perform the Migration for more details.
2. Remove the data directories and WAL directory on the unwanted masters. This is a precaution to ensure that they cannot start up again and interfere with the new multi-master deployment.
3. Modify the value of the `master_addresses` configuration parameter for the masters of the new multi-master deployment. See Kudu Configuration docs for the steps to modify a configuration parameter. If migrating to a single-master deployment, the `master_addresses` flag should be omitted entirely.
4. Start all of the masters that were not removed.

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

5. Modify the value of the `tserver_master_addrs` configuration parameter for the tablet servers to remove any unwanted masters. See Kudu Configuration docs for the steps to modify a configuration parameter.
6. Start all of 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. For more details, see Monitoring Cluster Health with ksck on page 32.

Changing Master Hostnames

When replacing dead masters, use DNS aliases to prevent long maintenance windows. If the cluster was set up without aliases, change the host names as described in this section.

Prepare for Hostname Changes

To prepare to change a hostname:

1. Establish a maintenance window during which the Kudu cluster will be unavailable. One hour should be sufficient.
2. On the **Masters** page in Kudu Web UI, note the UUID and RPC address of each master.
3. Stop all the Kudu processes in the cluster.
4. Set up the new hostnames to point to the masters and verify all servers and clients properly resolve them.

Perform Hostname Changes

To change hostnames:

1. Rewrite each master's Raft configuration with the following command, executed on each master host:

```
$ sudo -u kudu kudu local_replica cmeta rewrite_raft_config --fs_wal_dir=<master_wal_dir>
 [--fs_data_dirs=<master_data_dir>] 00000000000000000000000000000000 <all_masters>
```

For example:

```
$ sudo -u kudu kudu local_replica cmeta rewrite_raft_config
--fs_wal_dir=/data/kudu/master/wal --fs_data_dirs=/data/kudu/master/data
00000000000000000000000000000000 4aab798a69e94fab8d77069edff28ce0:new-master-name-1:7051
 f5624e05f40649b79a757629a69d061e:new-master-name-2:7051
988d8ac6530f426cbe180be5ba52033d:new-master-name-3:7051
```

2. Update the master address:
   - In an environment not managed by Cloudera Manager, change the `gflag` file of the masters so the `master_addresses` parameter reflects the new hostnames.
   - In an environment managed by Cloudera Manager, specify the new hostname in the **Master Address (server.address)** field on each Kudu role.
3. Change the `gflag` file of the tablet servers to update the `tserver_master_addrs` parameter with the new hostnames. In an environment managed by Cloudera Manager, this step is not needeed.
4. Start the masters.
5. To verify that all masters are working properly, perform the following sanity checks:
   a. In each master's Web UI, click **Masters** on the Status Pages. All of the masters should be listed there with one master in the **LEADER** role field and the others in the **FOLLOWER** role field. The contents of **Masters** on all master should be the same.
   b. Run the below command to verify all masters are up and listening. The UUIDs are the same and belong to the same master as before the hostname change:

```
$ sudo -u kudu kudu master list
new-master-name-1:7051,new-master-name-2:7051,new-master-name-3:7051
```

6. Start all of the tablet servers.
7. Run a Kudu system check (`ksck`) on the cluster using the `kudu` command line tool. See Monitoring Cluster Health with ksck on page 32 for more details. After startup, some tablets may be unavailable as it takes some time to initialize all of them.
8. If you have Kudu tables that are accessed from Impala, update the HMS database manually in the underlying database that provides the storage for HMS.
   a. The following is an example SQL statement you run in the HMS database:

```
UPDATE TABLE_PARAMSSET PARAM_VALUE =
'new-master-name-1:7051,new-master-name-2:7051,new-master-name-3:7051'
WHERE PARAM_KEY = 'kudu.master_addresses'
AND PARAM_VALUE = 'master-1:7051,master-2:7051,master-3:7051';
```

   b. In `impala-shell`, run:

```
INVALIDATE METADATA;
```

   c. Verify updating the metadata worked by running a simple `SELECT` query on a Kudu-backed Impala table.

Monitoring Cluster Health with ksck

The `kudu` CLI includes a tool called `ksck` which can be used for monitoring cluster health and data integrity. `ksck` will identify issues such as under-replicated tablets, unreachable tablet servers, or tablets without a leader.

`ksck` should be run from the command line, and requires you to specify the complete list of Kudu master addresses:

```
$ sudo -u kudu kudu cluster ksck
master-01.example.com,master-02.example.com,master-03.example.com
```

To see the full list of the options available with `ksck`, either use the `--help` flag or see [Kudu command line reference documentation].

If the cluster is healthy, `ksck` will print a success message, and return a zero (success) exit status.

```
Connected to the Master
Fetched info from all 1 Tablet Servers
Table IntegrationTestBigLinkedList is HEALTHY (1 tablet(s) checked)

The metadata for 1 table(s) is HEALTHY
OK
```

If the cluster is unhealthy, for instance if a tablet server process has stopped, `ksck` will report the issue(s) and return a non-zero exit status:

```
Connected to the Master
WARNING: Unable to connect to Tablet Server 8a0b66a756014def82760a09946d1fce
(tserver-01.example.com:7050): Network error: could not send Ping RPC to server: Client
 connection negotiation failed: client connection to 192.168.0.2:7050: connect: Connection
 refused (error 61)
WARNING: Fetched info from 0 Tablet Servers, 1 weren't reachable
Tablet ce3c2d27010d4253949a989b9d9bf43c of table 'IntegrationTestBigLinkedList'
is unavailable: 1 replica(s) not RUNNING
  8a0b66a756014def82760a09946d1fce (tserver-01.example.com:7050): TS unavailable [LEADER]

  Table IntegrationTestBigLinkedList has 1 unavailable tablet(s)

  : 1 out of 1 table(s) are not in a healthy state
  ==================
  Errors:
  ==================
  error fetching info from tablet servers: Network error: Not all Tablet Servers are
reachable
  table consistency check error: Corruption: 1 table(s) are bad

  FAILED
  Runtime error: ksck discovered errors
```

To verify data integrity, the optional `--checksum-scan` flag can be set, which will ensure that the cluster has consistent data by scanning each tablet replica and comparing results. The `--tables` and `--tablets` flags can be used to limit the scope of the checksum scan to specific tables or tablets, respectively.

For example, use the following command to check the integrity of data in the `IntegrationTestBigLinkedList` table:

```
$ sudo -k kudu kudu cluster ksck --checksum-scan --tables IntegrationTestBigLinkedList
 master-01.example.com,master-02.example.com,master-03.example.com
```

Bringing a Tablet That Has Lost a Majority of Replicas Back Online

If a tablet has permanently lost a majority of its replicas, it cannot recover automatically and operator intervention is required. The steps below may cause recent edits to the tablet to be lost, potentially resulting in permanent data loss. Only attempt the procedure below if it is impossible to bring a majority back online.

Suppose a tablet has lost a majority of its replicas. The first step in diagnosing and fixing the problem is to examine the tablet's state using `ksck`:

```
$ sudo -u kudu kudu cluster ksck --tablets=e822cab6c0584bc0858219d1539a17e6
master-00,master-01,master-02
Connected to the Master
Fetched info from all 5 Tablet Servers
Tablet e822cab6c0584bc0858219d1539a17e6 of table 'my_table' is unavailable: 2 replica(s)
 not RUNNING
  638a20403e3e4ae3b55d4d07d920e6de (tserver-00:7150): RUNNING
  9a56fa85a38a4edc99c6229cba68aeaa (tserver-01:7150): bad state
    State:       FAILED
    Data state:  TABLET_DATA_READY
```

```
    Last status: <failure message>
  c311fef7708a4cf9bb11a3e4cbcaab8c (tserver-02:7150): bad state
    State:       FAILED
    Data state:  TABLET_DATA_READY
    Last status: <failure message>
```

This output shows that, for tablet `e822cab6c0584bc0858219d1539a17e6`, the two tablet replicas on `tserver-01` and `tserver-02` failed. The remaining replica is not the leader, so the leader replica failed as well. This means the chance of data loss is higher since the remaining replica on `tserver-00` may have been lagging. In general, to accept the potential data loss and restore the tablet from the remaining replicas, divide the tablet replicas into two groups:

1. Healthy replicas: Those in `RUNNING` state as reported by `ksck`
2. Unhealthy replicas

For example, in the above `ksck` output, the replica on tablet server `tserver-00` is healthy while the replicas on `tserver-01` and `tserver-02` are unhealthy. On each tablet server with a healthy replica, alter the consensus configuration to remove unhealthy replicas. In the typical case of 1 out of 3 surviving replicas, there will be only one healthy replica, so the consensus configuration will be rewritten to include only the healthy replica.

```
$ sudo -u kudu kudu remote_replica unsafe_change_config tserver-00:7150 <tablet-id>
<tserver-00-uuid>
```

where `<tablet-id>` is `e822cab6c0584bc0858219d1539a17e6` and `<tserver-00-uuid>` is the uuid of `tserver-00`, `638a20403e3e4ae3b55d4d07d920e6de`.

Once the healthy replicas' consensus configurations have been forced to exclude the unhealthy replicas, the healthy replicas will be able to elect a leader. The tablet will become available for writes though it will still be under-replicated. Shortly after the tablet becomes available, the leader master will notice that it is under-replicated, and will cause the tablet to re-replicate until the proper replication factor is restored. The unhealthy replicas will be tombstoned by the master, causing their remaining data to be deleted.

Rebuilding a Kudu Filesystem Layout

Kudu does not allow removing directories or changing the write-ahead-log (WAL) or metadata directories. To start a server with such directory configuration changes, the WAL and data directories on the server must be deleted and rebuilt, destroying the copy of the data for each tablet replica hosted on the local server. Kudu will automatically re-replicate tablet replicas removed in this way, provided the replication factor is at least three and all other servers are online and healthy.

Note: These steps use a tablet server as an example, but the steps are the same for Kudu master servers.

1. The first step to rebuilding a server with a new directory configuration is emptying all of the server's existing directories. For example, if a tablet server is configured with `--fs_wal_dir=/data/0/kudu-tserver-wal` and `--fs_data_dirs=/data/1/kudu-tserver,/data/2/kudu-tserver`, the following commands will remove the contents in the write-ahead-log (WAL) directory and data directories:

```
$ rm -rf /data/0/kudu-tserver-wal/* /data/1/kudu-tserver/* /data/2/kudu-tserver/*
```

2. If using Cloudera Manager, update the configurations for the rebuilt server to include only the desired directories. Make sure to only update the configurations of servers to which changes were applied rather than of the entire Kudu service.
3. After the WAL and data directories are deleted, the server process can be started with the new directory configuration. Kudu will create the appropriate sub-directories when starting up.

Physical Backups of an Entire Node

Kudu does not yet provide any built-in backup and restore functionality. However, it is possible to create a physical backup of a Kudu node, either tablet server or master, and restore it later.

1. Stop all Kudu processes in the cluster. This prevents the tablets on the backed up node from being rereplicated elsewhere unnecessarily.
2. If creating a backup, make a copy of the WAL, metadata, and data directories on each node to be backed up. It is important that this copy preserve all file attributes as well as sparseness.
3. If restoring from a backup, delete the existing WAL, metadata, and data directories, then restore the backup via move or copy. As with creating a backup, it is important that the restore preserve all file attributes and sparseness.
4. Start all Kudu processes in the cluster.

Scaling Storage on Kudu Master and Tablet Servers in the Cloud

If you find that the size of your Kudu cloud deployment has exceeded previous expectations, or you simply wish to allocate more storage to Kudu, use the following set of high-level steps as a guide to increasing storage on your Kudu master or tablet server hosts. You must work with your cluster's Hadoop administrators and the system administrators to complete this process. Replace the file paths in the following steps to those relevant to your setup.

1. Run a consistency check on the cluster hosts. For instructions, see Monitoring Cluster Health with ksck on page 32.
2. On all Kudu hosts, create a new file system with the storage capacity you require. For example, `/new/data/dir`.
3. Shutdown cluster services. For a cluster managed by Cloudera Manager cluster, see Starting and Stopping a Cluster.
4. Copy the contents of your existing data directory, `/current/data/dir`, to the new filesystem at `/new/data/dir`.
5. Move your existing data directory, `/current/data/dir`, to a separate temporary location such as `/tmp/data/dir`.
6. Create a new `/current/data/dir` directory.

```
mkdir /current/data/dir
```

7. Mount `/new/data/dir` as `/current/data/dir`. Make changes to `fstab` as needed.
8. Perform steps 4-7 on all Kudu hosts.
9. Startup cluster services. For a cluster managed by Cloudera Manager cluster, see Starting and Stopping a Cluster.
10. Run a consistency check on the cluster hosts. For instructions, see Monitoring Cluster Health with ksck on page 32.
11. After 10 days, if everything is in working order on all the hosts, get approval from the Hadoop administrators to remove the `/backup/data/dir` directory.

Managing Kudu Using Cloudera Manager

This topic describes the tasks you can perform to manage the Kudu service using Cloudera Manager. You can use the Kudu service to upgrade the Kudu service, start and stop the Kudu service, monitor operations, and configure the Kudu master and tablet servers, among other tasks. Depending on your deployment, there are several different configuration settings you may need to modify.

For detailed information about Apache Kudu, view the Apache Kudu Guide.

Installing and Upgrading the Kudu Service

You can install Kudu through the Cloudera Manager installation wizard, using either parcels or packages. For instructions, see Installing Kudu.

For instructions on upgrading Kudu using parcels or packages, see Upgrading Kudu.

Enabling Core Dump for the Kudu Service

If Kudu crashes, you can use Cloudera Manager to generate a core dump to get more information about the crash.

1. Go to the Kudu service.
2. Click the **Configuration** tab.
3. Search for `core dump`.
4. Check the checkbox for the **Enable Core Dump** property.
5. (Optional) Unless otherwise configured, the dump file is generated in the default core dump directory, `/var/log/kudu`, for both the Kudu master and the tablet servers.
   - To configure a different dump directory for the Kudu master, modify the value of the **Kudu Master Core Dump Directory** property.
   - To configure a different dump directory for the Kudu tablet servers, modify the value of the **Kudu Tablet Server Core Dump Directory** property.
6. Click **Save Changes**.

Verifying the Impala Dependency on Kudu

In a Cloudera Manager deployment, once the Kudu service is installed, Impala will automatically identify the Kudu Master. However, if your Impala queries don't work as expected, use the following steps to make sure that the Impala service is set to be dependent on Kudu.

1. Go to the Impala service.
2. Click the **Configuration** tab.
3. Search for `kudu`.
4. Make sure the **Kudu Service** property is set to the right Kudu service.
5. Click **Save Changes**.

Using the Charts Library with the Kudu Service

By default, the **Status** tab for the Kudu service displays a dashboard containing a limited set of charts. For details on the terminology used in these charts, and instructions on how to query for time-series data, display chart details, and edit charts, see Charting Time-Series Data.

The Kudu service's Charts Library tab also displays a dashboard containing a much larger set of charts, organized by categories such as process charts, host charts, CPU charts, and so on, depending on the entity (service, role, or host) that you are viewing. You can use these charts to keep track of disk space usage, the rate at which data is being inserted/modified in Kudu across all tables, or any critical cluster events. You can also use them to keep track of individual tables. For example, to find out how much space a Kudu table is using on disk:

1. Go to the Kudu service and navigate to the **Charts Library** tab.
2. On the left-hand side menu, click **Tables** to display the list of tables currently stored in Kudu.
3. Click on a table name to view the default dashboard for that table. The **Total Tablet Size On Disk Across Kudu Replicas** chart displays the total size of the table on disk using a time-series chart.

   Hovering with your mouse over the line on the chart opens a small pop-up window that displays information about that data point. Click the data stream within the chart to display a larger pop-up window that includes additional information for the table at the point in time where the mouse was clicked.

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

The 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.

Building the Java Client

Requirements

- JDK 7
- Apache Maven 3.x
- `protoc` 2.6 or newer installed in your path, or built from the `thirdparty/` directory. Run the following commands to build `protoc` from the third-party dependencies:

```
thirdparty/download-thirdparty.sh
thirdparty/build-thirdparty.sh protobuf
```

To build the Java client, clone the Kudu Git repository, change to the `java` directory, and issue the following command:

```
$ mvn install -DskipTests
```

For more information about building the Java API, as well as Eclipse integration, see `java/README.md`.

Kudu Python Client

The Kudu Python client provides a Python friendly interface to the C++ client API.

To install and use Kudu Python client, you need to install the Kudu C++ client libraries and headers. See Kudu Installation for installing Kudu C++ client.

To install the Kudu Python client:

1. Install Cython: `sudo pip install cython`
2. Downloaded the Kudu Python client from kudu-python: `kudu-python-1.2.0.tar.gz`
3. Install kudu-python: `sudo pip install kudu-python`

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 with Kudu on page 43 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:

Spark 1.x - Use the `kudu-spark_2.10` artifact if you are using Spark 1.x with Scala 2.10:

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

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

```
spark2-shell --packages org.apache.kudu:kudu-spark2_2.11:1.4.0
```

Then import kudu-spark and create a dataframe as demonstrated in the following sample code. In the following example, replace `<kudu.master>` with the actual hostname of the host running a Kudu master service, and *<kudu_table>* with the name of a pre-existing table in Kudu.

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

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

// Query <kudu_table> 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", sqlContext.sparkContext)

// 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")
```

Using Spark with a Secure Kudu cluster

The Kudu-Spark integration is able to operate on secure Kudu clusters which have authentication and encryption enabled, but the submitter of the Spark job must provide the proper credentials. For Spark jobs using the default 'client' deploy mode, the submitting user must have an active Kerberos ticket granted through `kinit`. For Spark jobs using the 'cluster' deploy mode, a Kerberos principal name and keytab location must be provided through the `--principal` and `--keytab` arguments to `spark2-submit`.

Spark Integration Known Issues and Limitations

- Spark 2.2 (and higher) requires Java 8 at runtime even though Kudu Spark 2.x integration is Java 7 compatible. Spark 2.2 is the default dependency version as of Kudu 1.5.0.
- 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 must not be used with SparkSQL. Columns can be renamed in Kudu to work around this issue.
- `<>` and `OR`predicates are not pushed to Kudu, and instead will be evaluated by the Spark task. Only `LIKE` predicates with a suffix wildcard are pushed to Kudu. This means `LIKE "FOO%"` will be pushed, but `LIKE "FOO%BAR"` won't.
- Kudu does not support all the types supported by Spark SQL. For example, `Date`, `Decimal`, and complex types are not supported on Kudu.
- Kudu tables can only be registered as temporary tables in SparkSQL.
- Kudu tables cannot be queried using HiveContext.

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 with Kudu

Apache Kudu has tight integration with Apache Impala, 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 higher.
- The syntax described in this topic is specific to Impala included in CDH 5.10 and higher, and will not work on previous versions. If you are using an lower version of Impala (including the `IMPALA_KUDU` releases previously available), upgrade to CDH 5.10 or higher.

  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 higher) 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.

The prefix `impala::` and the Impala database name are appended to the underlying Kudu table name:
`impala::<database>.<table>`

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::impala_kudu.my_first_table`. This also applies to `INSERT`, `UPDATE`, `DELETE`, and `DROP` statements.

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 `impala::database_name.table_name`. The prefix is always `impala::`, and the database name and table name follow, separated by a dot.

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 examples in this section 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 45.

Note: In Impala included in CDH 5.13 and higher, the `PARTITION BY` clause is optional for Kudu tables. If the clause is omitted, Impala automatically constructs a single partition that is not connected to any column. Because such a table cannot take advantage of Kudu features for parallelized queries and query optimizations, omitting the `PARTITION BY` clause is only appropriate for small lookup tables.

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;
```

By default, Kudu tables created through Impala use a tablet replication factor of 3. To specify the replication factor for a Kudu table, add a `TBLPROPERTIES` clause to the `CREATE TABLE` statement as shown below where *n* is the replication factor you want to use:

```
TBLPROPERTIES ('kudu.num_tablet_replicas' = 'n')
```

A replication factor must be an odd number.

Changing the `kudu.num_tablet_replicas` table property using the `ALTER TABLE` currently has no effect.

The Impala SQL Reference CREATE TABLE topic has more details and examples.

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.

Note: In Impala included in CDH 5.13 and higher, the `PARTITION BY` clause is optional for Kudu tables. If the clause is omitted, Impala automatically constructs a single partition that is not connected to any column. Because such a table cannot take advantage of Kudu features for parallelized queries and query optimizations, omitting the `PARTITION BY` clause is only appropriate for small lookup tables.

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 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 47 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 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: See Range Partitioning on page 63 for the caveats of non-covering range partitions.

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 52). 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 49.

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 Example

The following example demonstrates how the `UPSERT` statement works. We start by creating two tables, `foo1` and `foo2`.

```
CREATE TABLE foo1 (
 id INT PRIMARY KEY,
 col1 STRING,
 col2 STRING
)
PARTITION BY HASH(id) PARTITIONS 3
STORED AS KUDU;
```

```
CREATE TABLE foo2 (
 id INT PRIMARY KEY,
 col1 STRING,
 col2 STRING
)
PARTITION BY HASH(id) PARTITIONS 3
STORED AS KUDU;
```

Populate `foo1` and `foo2` using the following `INSERT` statements. For `foo2`, we leave column `col2` with NULL values to be upserted later:

```
INSERT INTO foo1 VALUES (1, "hi", "alice");
```

```
INSERT INTO foo2 select id, col1, NULL from foo1;
```

The contents of `foo2` will be:

```
SELECT * FROM foo2;
...
+----+------+------+
| id | col1 | col2 |
+----+------+------+
| 1  | hi   | NULL |
+----+------+------+
Fetched 1 row(s) in 0.15s
```

Now use the `UPSERT` command to now replace the NULL values in foo2 with the actual values from foo1.

```
UPSERT INTO foo2 (id, col2) select id, col2 from foo1;
```

```
SELECT * FROM foo2;
...
+----+------+-------+
| id | col1 | col2  |
+----+------+-------+
| 1  | hi   | alice |
+----+------+-------+
Fetched 1 row(s) in 0.15s
```

Altering a Table

You can the `ALTER TABLE` statement to change the default value, encoding, compression, or block size of existing columns in a Kudu table.

The Impala SQL Reference ALTER TABLE includes a **Kudu Considerations** section with examples and a list of constraints relevant to altering a Kudu table in Impala.

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 49.

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 52.

```
DROP TABLE my_first_table;
```

Security Considerations

Kudu 1.3 (and higher) includes security features that allow Kudu clusters to be hardened against access from unauthorized users. Kudu uses strong authentication with Kerberos, while communication between Kudu clients and servers can now be encrypted with TLS. Kudu also allows you to use HTTPS encryption to connect to the web UI. These features should work seamlessly in Impala as long as Impala's user is given permission to access Kudu.

For instructions on how to configure a secure Kudu cluster, see Kudu Security on page 55.

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 `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 cannot be used as an external table in Impala. Columns can be renamed in Kudu to work around this issue.
- `!=` 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.

Kudu Security

Kudu includes security features that allow Kudu clusters to be hardened against access from unauthorized users. Kudu uses strong authentication with Kerberos, while communication between Kudu clients and servers can now be encrypted with TLS. Kudu also allows you to use HTTPS encryption to connect to the web UI.

The rest of this topic describes the security capabilities of Apache Kudu and how to configure a secure Kudu cluster. Currently, there are a few known limitations in Kudu security that might impact your cluster. For the list, see Security Limitations on page 18.

Kudu Authentication with Kerberos

Kudu can be configured to enforce secure authentication among servers, and between clients and servers. Authentication prevents untrusted actors from gaining access to Kudu, and securely identifies connecting users or services for authorization checks. Authentication in Kudu is designed to interoperate with other secure Hadoop components by utilizing Kerberos.

Configure authentication on Kudu servers using the `--rpc-authentication` flag, which can be set to one of the following options:

- `required` - Kudu will reject connections from clients and servers who lack authentication credentials.
- `optional` - Kudu will attempt to use strong authentication, but will allow unauthenticated connections.
- `disabled` - Kudu will only allow unauthenticated connections.

By default, the flag is set to `optional`. To secure your cluster, set `--rpc-authentication` to `required`.

Internal Private Key Infrastructure (PKI)

Kudu uses an internal PKI to issue X.509 certificates to servers in the cluster. Connections between peers who have both obtained certificates will use TLS for authentication. In such cases, neither peer needs to contact the Kerberos KDC.

X.509 certificates are only used for internal communication among Kudu servers, and between Kudu clients and servers. These certificates are never presented in a public facing protocol. By using internally-issued certificates, Kudu offers strong authentication which scales to huge clusters, and allows TLS encryption to be used without requiring you to manually deploy certificates on every node.

Authentication Tokens

After authenticating to a secure cluster, the Kudu client will automatically request an authentication token from the Kudu master. An authentication token encapsulates the identity of the authenticated user and carries the Kudu master's RSA signature so that its authenticity can be verified. This token will be used to authenticate subsequent connections. By default, authentication tokens are only valid for seven days, so that even if a token were compromised, it cannot be used indefinitely. For the most part, authentication tokens should be completely transparent to users. By using authentication tokens, Kudu is able to take advantage of strong authentication, without paying the scalability cost of communicating with a central authority for every connection.

When used with distributed compute frameworks such as Apache Spark, authentication tokens can simplify configuration and improve security. For example, the Kudu Spark connector will automatically retrieve an authentication token during the planning stage, and distribute the token to tasks. This allows Spark to work against a secure Kudu cluster where only the planner node has Kerberos credentials.

Client Authentication to Secure Kudu Clusters

Users running client Kudu applications must first run the `kinit` command to obtain a Kerberos ticket-granting ticket. For example:

```
$ kinit admin@EXAMPLE-REALM.COM
```

Once authenticated, you use the same client code to read from and write to Kudu servers with and without the Kerberos configuration.

Scalability

Kudu authentication is designed to scale to thousands of nodes, which means it must avoid unnecessary coordination with a central authentication authority (such as the Kerberos KDC) for each connection. Instead, Kudu servers and clients use Kerberos to establish initial trust with the Kudu master, and then use alternate credentials for subsequent connections. As described previously, the Kudu master issues internal X.509 certificates to tablet servers on startup, and temporary authentication tokens to clients on first contact.

Encryption

Kudu allows you to use TLS to encrypt all communications among servers, and between clients and servers. Configure TLS encryption on Kudu servers using the `--rpc-encryption` flag, which can be set to one of the following options:

- `required` - Kudu will reject unencrypted connections.
- `optional` - Kudu will attempt to use encryption, but will allow unencrypted connections.
- `disabled` - Kudu will not use encryption.

By default, the flag is set to `optional`. To secure your cluster, set `--rpc-encryption` to `required`.

Note: Kudu will automatically turn off encryption on local loopback connections, since traffic from these connections is never exposed externally. This allows locality-aware compute frameworks, such as Spark and Impala, to avoid encryption overhead, while still ensuring data confidentiality.

Coarse-grained Authorization

Kudu supports coarse-grained authorization checks for client requests based on the client's authenticated Kerberos principal (user or service). Access levels are granted based on whitelist-style Access Control Lists (ACLs), one for each level. Each ACL specifies a comma-separated list of users, or may be set to '`*`' to indicate that all authenticated users have access rights at the specified level.

The two levels of access which can be configured are:

- **Superuser** - Principals authorized as a superuser can perform certain administrative functions such as using the `kudu` command line tool to diagnose and repair cluster issues.
- **User** - Principals authorized as a user are able to access and modify all data in the Kudu cluster. This includes the ability to create, drop, and alter tables, as well as read, insert, update, and delete data. The default value for the User ACL is '`*`', which allows all users access to the cluster. However, if authentication is enabled, this will restrict access to only those users who are able to successfully authenticate using Kerberos. Unauthenticated users on the same network as the Kudu servers will be unable to access the cluster.

Note: Internally, Kudu has a third access level for the daemons themselves called **Service**. This is used to ensure that users cannot connect to the cluster and pose as tablet servers.

Web UI Encryption

The Kudu web UI can be configured to use secure HTTPS encryption by providing each server with TLS certificates. Use the `--webserver-certificate-file` and `--webserver-private-key-file` properties to specify the certificate and private key to be used for communication.

Alternatively, you can choose to completely disable the web UI by setting `--webserver-enabled` flag to `false` on the Kudu servers.

Web UI Redaction

To prevent sensitive data from being included in the web UI, all row data is redacted. Table metadata, such as table names, column names, and partitioning information is not redacted. Alternatively, you can choose to completely disable the web UI by setting the `--webserver-enabled` flag to `false` on the Kudu servers.

Note: Disabling the web UI will also disable REST endpoints such as `/metrics`. Monitoring systems rely on these endpoints to gather metrics data.

Log Redaction

To prevent sensitive data from being included in Kudu server logs, all row data will be redacted. You can turn off log redaction using the `--redact` flag.

Configuring a Secure Kudu Cluster using Cloudera Manager

Warning: If you are upgrading from Kudu 1.2.0 / CDH 5.10.x, you must upgrade both Kudu and CDH parcels (or packages) at the same time. If you upgrade Kudu but do not upgrade CDH, new Kudu features such as Security will not be available. Note that even though you might be able to see the updated configuration options for Kudu security in Cloudera Manager, configuring them will have no effect.

Use the following set of instructions to secure a Kudu cluster using Cloudera Manager:

Enabling Kerberos Authentication and RPC Encryption

Important: The following instructions assume you already have a secure Cloudera Manager cluster with Kerberos authentication enabled. If this is not the case, first secure your cluster using the steps described at Enabling Kerberos Authentication Using the Cloudera Manager Wizard.

To enable Kerberos authentication for Kudu:

1. Go to the **Kudu** service.
2. Click the **Configuration** tab.
3. Select **Category** > **Main**.
4. In the Search field, type **Kerberos** to show the relevant properties.
5. Edit the following properties according to your cluster configuration:

Field	Usage Notes
Kerberos Principal	Set to the default principal, `kudu`. Currently, Kudu does not support configuring a custom service principal for Kudu processes.

Field	Usage Notes
Enable Secure Authentication And Encryption	Select this checkbox to enable authentication and RPC encryption between all Kudu clients and servers, as well as between individual servers. Only enable this property after you have configured Kerberos.

6. Click **Save Changes**.
7. You will see an error message that tells you the Kudu keytab is missing. To generate the keytab, go to the top navigation bar and click **Administration** > **Security**.
8. Go to the **Kerberos Credentials** tab. On this page you will see a list of the existing Kerberos principals for services running on the cluster.
9. Click **Generate Missing Credentials**. Once the Generate Missing Credentials command has finished running, you will see the Kudu principal added to the list.

Configuring Coarse-grained Authorization with ACLs

1. Go to the **Kudu** service.
2. Click the **Configuration** tab.
3. Select **Category** > **Security**.
4. In the Search field, type **ACL** to show the relevant properties.
5. Edit the following properties according to your cluster configuration:

Field	Usage Notes
***Superuser* Access Control List**	Add a comma-separated list of superusers who can access the cluster. By default, this property is left blank. '*' indicates that all authenticated users will be given superuser access.
***User* Access Control List**	Add a comma-separated list of users who can access the cluster. By default, this property is set to '*'. The default value of `'*'` allows all users access to the cluster. However, if authentication is enabled, this will restrict access to only those users who are able to successfully authenticate using Kerberos. Unauthenticated users on the same network as the Kudu servers will be unable to access the cluster. Add the `impala` user to this list to allow Impala to query data in Kudu. You might choose to add any other relevant usernames if you want to give access to Spark Streaming jobs.

6. Click **Save Changes**.

Configuring HTTPS Encryption for the Kudu Master and Tablet Server Web UIs

Use the following steps to enable HTTPS for encrypted connections to the Kudu master and tablet server web UIs.

1. Go to the **Kudu** service.
2. Click the **Configuration** tab.
3. Select **Category** > **Security**.
4. In the Search field, type **TLS/SSL** to show the relevant properties.
5. Edit the following properties according to your cluster configuration:

Field	Usage Notes
Master TLS/SSL Server Private Key File (PEM Format)	Set to the path containing the Kudu master host's private key (PEM-format). This is used to enable TLS/SSL encryption (over HTTPS) for browser-based connections to the Kudu master web UI.

Field	Usage Notes
Tablet Server TLS/SSL Server Private Key File (PEM Format)	Set to the path containing the Kudu tablet server host's private key (PEM-format). This is used to enable TLS/SSL encryption (over HTTPS) for browser-based connections to Kudu tablet server web UIs.
Master TLS/SSL Server Certificate File (PEM Format)	Set to the path containing the signed certificate (PEM-format) for the Kudu master host's private key (set in Master TLS/SSL Server Private Key File). The certificate file can be created by concatenating all the appropriate root and intermediate certificates required to verify trust.
Tablet Server TLS/SSL Server Certificate File (PEM Format)	Set to the path containing the signed certificate (PEM-format) for the Kudu tablet server host's private key (set in Tablet Server TLS/SSL Server Private Key File). The certificate file can be created by concatenating all the appropriate root and intermediate certificates required to verify trust.
Enable TLS/SSL for Master Server	Enables HTTPS encryption on the Kudu master web UI.
Enable TLS/SSL for Tablet Server	Enables HTTPS encryption on the Kudu tablet server Web UIs.

6. Click **Save Changes**.

Configuring a Secure Kudu Cluster using the Command Line

Important: Follow these command-line instructions on systems that do not use Cloudera Manager. If you are using Cloudera Manager, see Configuring a Secure Kudu Cluster using Cloudera Manager on page 57.

The following configuration parameters should be set on all servers (master and tablet servers) to ensure that a Kudu cluster is secure:

```
# Connection Security
#--------------------
--rpc_authentication=required
--rpc_encryption=required
--keytab_file=<path-to-kerberos-keytab>

# Web UI Security
#--------------------
--webserver_certificate_file=<path-to-cert-pem>
--webserver_private_key_file=<path-to-key-pem>
# optional
--webserver_private_key_password_cmd=<password-cmd>

# If you prefer to disable the web UI entirely:
--webserver_enabled=false

# Coarse-grained authorization
#--------------------------------

# This example ACL setup allows the 'impala' user as well as the
# 'etl_service_account' principal access to all data in the
# Kudu cluster. The 'hadoopadmin' user is allowed to use administrative
# tooling. Note that by granting access to 'impala', other users
# may access data in Kudu via the Impala service subject to its own
# authorization rules.
--user_acl=impala,etl_service_account
--admin_acl=hadoopadmin
```

More information about these flags can be found in the configuration reference documentation.

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 uncompressed)
- binary (up to 64KB uncompressed)

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. Starting with Kudu 1.3, default encodings are specific to each column type.

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

Column Compression

Kudu allows per-column compression using the `LZ4`, `Snappy`, or `zlib` compression codecs.

By default, columns that are Bitshuffle-encoded are inherently compressed with the `LZ4` compression. Otherwise, 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 comprised of one or more columns. 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.

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.

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 in a clustered index. All rows within a tablet are sorted by its primary key.

When scanning Kudu rows, use equality or range predicates on primary key columns to efficiently find the rows.

Considerations for Backfill Inserts

This section discuss a primary key design consideration for timeseries use cases where the primary key is a timestamp, or the first column of the primary key is a timestamp.

Each time a row is inserted into a Kudu table, Kudu looks up the primary key in the primary key index storage to check whether that primary key is already present in the table. If the primary key exists in the table, a "duplicate key" error is returned. In the typical case where data is being inserted at the current time as it arrives from the data source, only a small range of primary keys are "hot". So, each of these "check for presence" operations is very fast. It hits the cached primary key storage in memory and doesn't require going to disk.

In the case when you load historical data, which is called "backfilling", from an offline data source, each row that is inserted is likely to hit a cold area of the primary key index which is not resident in memory and will cause one or more HDD disk seeks. For example, in a normal ingestion case where Kudu sustains a few million inserts per second, the "backfill" use case might sustain only a few thousand inserts per second.

To alleviate the performance issue during backfilling, consider the following options:

- Make the primary keys more compressible.

  For example, with the first column of a primary key being a random ID of 32-bytes, caching one billion primary keys would require at least 32 GB of RAM to stay in cache. If caching backfill primary keys from several days ago, you need to have several times 32 GB of memory. By changing the primary key to be more compressible, you increase the likelihood that the primary keys can fit in cache and thus reducing the amount of random disk I/Os.

- Use SSDs for storage as random seeks are orders of magnitude faster than spinning disks.
- Change the primary key structure such that the backfill writes hit a continuous range of primary keys.

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.

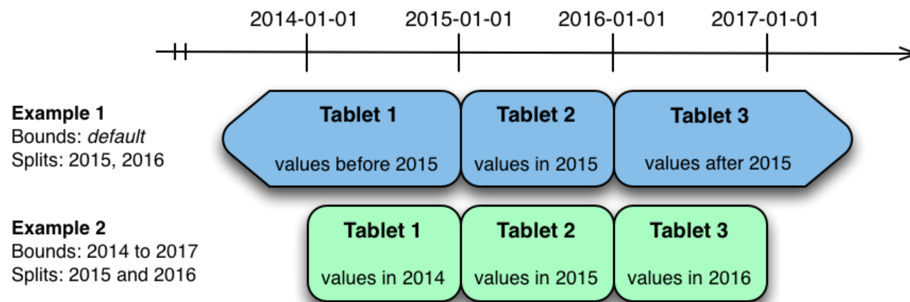
Partitioning Examples

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.



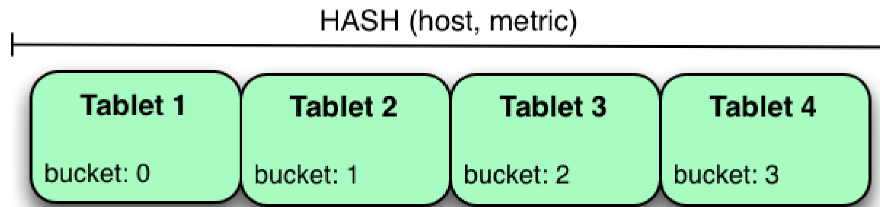
The image above shows the two ways the `metrics` table can be range partitioned on the `time` column. In the first example (in blue), the default range partition bounds are used, with splits at `2015-01-01` and `2016-01-01`. This results in three tablets: the first containing values before 2015, the 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.

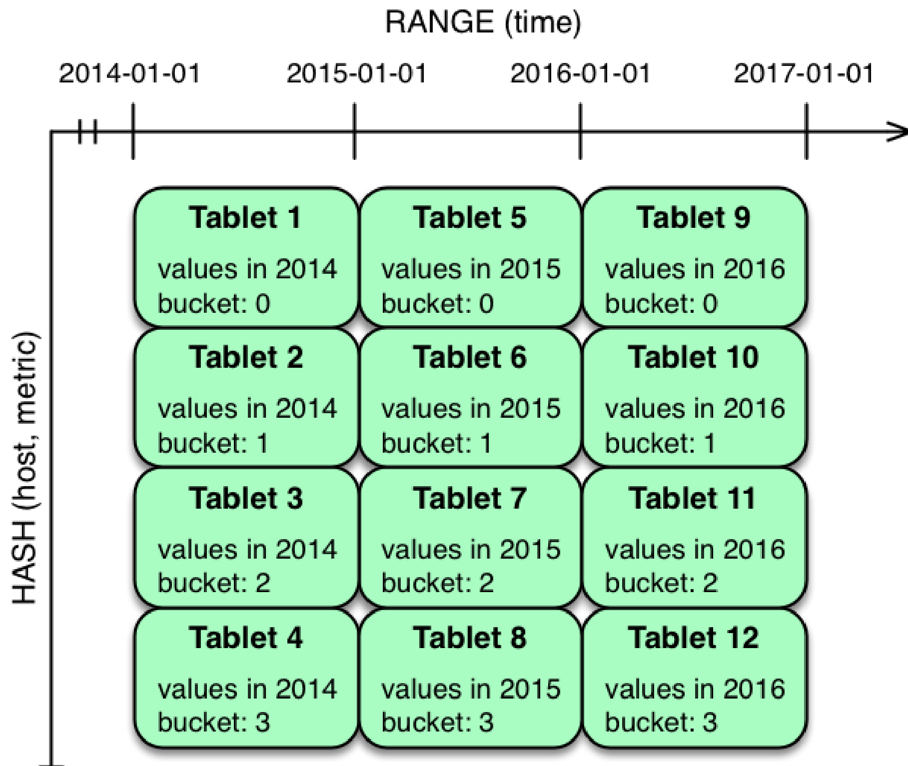
Hash and Range Partitioning

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 3: 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

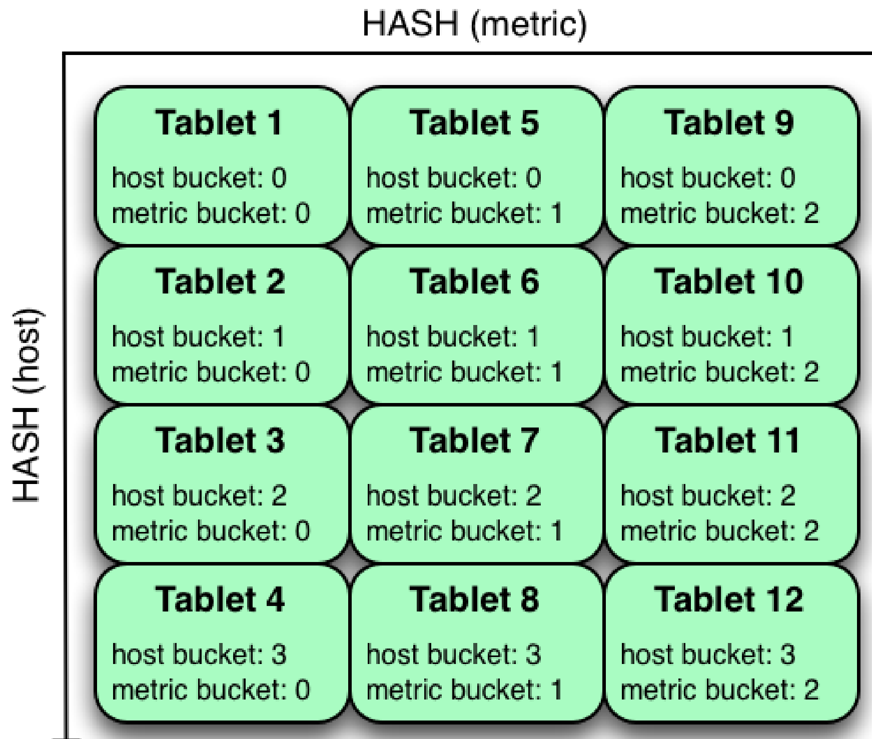
Hash partitioning is good at maximizing write throughput, while range partitioning avoids issues of unbounded tablet growth. Both strategies can take advantage of partition pruning to optimize scans in different scenarios. Using multilevel partitioning, it is possible to combine the two strategies in 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 primary key columns
- 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 Usage Limitations on page 15.

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 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#ReadMode(…)`

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 for better scan performance.

Note:

Kudu also provides replica selection API for you to choose at which replica the scan should be performed:

Java Client

Call `KuduScannerBuilder#replicaSelection(...)`

C++ Client

Call `KuduScanner::SetSelection(...)`

This API is a means to control locality and, in some cases, latency. The replica selection API has no effect on the consistency guarantees, which will hold no matter which replica is selected.

Known Issues and Limitations

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

Writes

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`

Reads (Scans)

- 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.

Apache Kudu Background Maintenance Tasks

Kudu relies on running background tasks for many important maintenance activities. These tasks include flushing data from memory to disk, compacting data to improve performance, freeing up disk space, and more.

Maintenance Manager

The maintenance manager schedules and runs background tasks. At any given point in time, the maintenance manager is prioritizing the next task based on improvements needed at that moment, such as relieving memory pressure, improving read performance, or freeing up disk space. The number of worker threads dedicated to running background tasks can be controlled by setting `--maintenance_manager_num_threads`.

With Kudu 1.4, the maintenance manager features improved utilization of the configured maintenance threads. Previously, maintenance work would only be scheduled a maximum of 4 times per second, but now maintenance work will be scheduled immediately whenever any configured thread is available. Make sure that the `--maintenance_manager_num_threads` property is set to at most a 1:3 ratio for Maintenance Manager threads to the number of data directories (for spinning disks). This will improve the throughput of write-heavy workloads.

Flushing Data to Disk

Flushing data from memory to disk relieves memory pressure and can improve read performance by switching from a write-optimized, row-oriented in-memory format in the `MemRowSet`, to a read-optimized, column-oriented format on disk.

Background tasks that flush data include `FlushMRSOp` and `FlushDeltaMemStoresOp`. The metrics associated with these operations have the prefix `flush_mrs` and `flush_dms`, respectively.

With Kudu 1.4, the maintenance manager aggressively schedules flushes of in-memory data when memory consumption crosses 60 percent of the configured process-wide memory limit. The backpressure mechanism which begins to throttle client writes was also adjusted to not begin throttling until memory consumption reaches 80 percent of the configured limit. These two changes together result in improved write throughput, more consistent latency, and fewer timeouts due to memory exhaustion.

Compacting On-disk Data

Kudu constantly performs several compaction tasks in order to maintain consistent read and write performance over time.

- A merging compaction, which combines multiple `DiskRowSets` together into a single `DiskRowSet`, is run by `CompactRowSetsOp`.
- Kudu also runs two types of delta store compaction operations: `MinorDeltaCompactionOp` and `MajorDeltaCompactionOp`.

  For more information on what these compaction operations do, see the Kudu Tablet design document.

The metrics associated with these tasks have the prefix `compact_rs`, `delta_minor_compact_rs`, and `delta_major_compact_rs`, respectively.

Write-ahead Log Garbage Collection

Kudu maintains a write-ahead log (WAL) per tablet that is split into discrete fixed-size segments. A tablet periodically rolls the WAL to a new log segment when the active segment reaches a size threshold (configured by the `--log_segment_size_mb` property). In order to save disk space and decrease startup time, a background task called `LogGCOp` attempts to garbage-collect (GC) old WAL segments by deleting them from disk once it is determined that they are no longer needed by the local node for durability.

The metrics associated with this background task have the prefix `log_gc`.

Tablet History Garbage Collection and the Ancient History Mark

Kudu uses a multiversion concurrency control (MVCC) mechanism to ensure that snapshot scans can proceed isolated from new changes to a table. Therefore, periodically, old historical data should be garbage-collected (removed) to free up disk space. While Kudu never removes rows or data that are visible in the latest version of the data, Kudu does remove records of old changes that are no longer visible.

The specific threshold in time (in the past) beyond which historical MVCC data becomes inaccessible and is free to be deleted is called the *ancient history mark* (AHM). The AHM can be configured by setting the `--tablet_history_max_age_sec` property.

There are two background tasks that remove historical MVCC data older than the AHM:

- The one that runs the merging compaction, called `CompactRowSetsOp` (see above).
- A separate background task deletes old undo delta blocks, called `UndoDeltaBlockGCOp`. Running `UndoDeltaBlockGCOp` reduces disk space usage in all workloads, but particularly in those with a higher volume of updates or upserts. The metrics associated with this background task have the prefix `undo_delta_block`.

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

Errors During Hole Punching Test

Kudu requires hole punching capabilities in order to be efficient. Hole punching support depends upon your operation system kernel version and local filesystem implementation.

- 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 and xfs filesystems support hole punching. Older versions that do not support hole punching will 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.
```

Note:

ext4 mountpoints may actually be backed by ext2 or ext3 formatted devices, which do not support hole punching. The hole punching test will fail when run on such filesystems. There are several different ways to determine whether an ext4 mountpoint is backed by an ext2, ext3, or ext4 formatted device; see this Stack Exchange post for details.

Without hole punching support, the log block manager is unsafe to use. It won't ever delete blocks, and will consume ever more space on disk.

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. The file block manager does not scale as well as the log block manager.

NTP Clock Synchronization Issues

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`.

Disk Space Usage

When using the log block manager (the default on Linux), Kudu uses sparse files to store data. A sparse file has a different apparent size than the actual amount of disk space it uses. This means that some tools may inaccurately report the disk space used by Kudu. For example, the size listed by `ls -l` does not accurately reflect the disk space used by Kudu data files:

```
$ ls -lh /data/kudu/tserver/data
total 117M
-rw------- 1 kudu kudu 160M Mar 26 19:37 0b9807b8b17d48a6a7d5b16bf4ac4e6d.data
-rw------- 1 kudu kudu 4.4K Mar 26 19:37 0b9807b8b17d48a6a7d5b16bf4ac4e6d.metadata
```

```
-rw------- 1 kudu kudu  32M Mar 26 19:37 2f26eeacc7e04b65a009e2c9a2a8bd20.data
-rw------- 1 kudu kudu 4.3K Mar 26 19:37 2f26eeacc7e04b65a009e2c9a2a8bd20.metadata
-rw------- 1 kudu kudu 672M Mar 26 19:37 30a2dd2cd3554d8a9613f588a8d136ff.data
-rw------- 1 kudu kudu 4.4K Mar 26 19:37 30a2dd2cd3554d8a9613f588a8d136ff.metadata
-rw------- 1 kudu kudu  32M Mar 26 19:37 7434c83c5ec74ae6af5974e4909cbf82.data
-rw------- 1 kudu kudu 4.3K Mar 26 19:37 7434c83c5ec74ae6af5974e4909cbf82.metadata
-rw------- 1 kudu kudu 672M Mar 26 19:37 772d070347a04f9f8ad2ad3241440090.data
-rw------- 1 kudu kudu 4.4K Mar 26 19:37 772d070347a04f9f8ad2ad3241440090.metadata
-rw------- 1 kudu kudu 160M Mar 26 19:37 86e50a95531f46b6a79e671e6f5f4151.data
-rw------- 1 kudu kudu 4.4K Mar 26 19:37 86e50a95531f46b6a79e671e6f5f4151.metadata
-rw------- 1 kudu kudu  687 Mar 26 19:26 block_manager_instance
```

Notice that the total size reported is 117MiB, while the first file's size is listed as 160MiB. Adding the `-s` option to `ls` will cause `ls` to output the file's disk space usage.

The `du` and `df` utilities report the actual disk space usage by default.

```
$ du -h /data/kudu/tserver/data118M   /data/kudu/tserver/data
```

The apparent size can be shown with the `--apparent-size` flag to `du`.

```
$ du -h --apparent-size /data/kudu/tserver/data1.7G  /data/kudu/tserver/data
```

Reporting Kudu Crashes Using Breakpad

Kudu uses the Google Breakpad library to generate a minidump whenever Kudu experiences a crash. A minidump file contains important debugging information about the process that crashed, including shared libraries loaded and their versions, a list of threads running at the time of the crash, the state of the processor registers and a copy of the stack memory for each thread, and CPU and operating system version information. These minidumps are typically only a few MB in size and are generated even if core dump generation is disabled. Currently, generating minidumps is only possible on Linux deployments.

By default, Kudu stores its minidumps in a subdirectory of the configured glog directory called `minidumps`. This location can be customized by setting the `--minidump_path` flag. Kudu will retain only a certain number of minidumps before deleting the older ones, in an effort to avoid filling up the disk with minidump files. The maximum number of minidumps that will be retained can be controlled by setting the `--max_minidumps` gflag.

Minidumps contain information specific to the binary that created them and are therefore not useful without access to the exact binary that crashed, or a very similar binary.

Kudu developers can access the minidump tools in their development environment because they are installed as part of the Kudu thirdparty build. They can be found in the Kudu development environment under `uninstrumented/bin`. For example, `thirdparty/installed/uninstrumented/bin/minidump-2-core`.

If minidumps are enabled, it is possible to force Kudu to create a minidump without killing the process. To do that, send a `USR1` signal to the `kudu-tserver` or `kudu-master` process. For example:

```
sudo pkill -USR1 kudu-tserver
```

Viewing the minidump stack trace with the GNU Debugger

Although a minidump contains no heap information, it does contain thread and stack information. You can convert a minidump to a core file to view it with GDB.

To convert the minidump (`.dmp` file) to a core file:

```
minidump-2-core -o 02cb4a97-ee37-6454-73a9d9cb-590c7dde.core \
02cb4a97-ee37-6454-73a9d9cb-590c7dde.dmp
```

To view the core file with GDB (on a parcel deployment):

```
gdb /opt/cloudera/parcels/KUDU/lib/kudu/sbin-release/kudu-master \
-s /opt/cloudera/parcels/KUDU/lib/debug/usr/lib/kudu/sbin-release/kudu-master.debug \
02cb4a97-ee37-6454-73a9d9cb-590c7dde.core
```

For more information, see Getting started with Breakpad and Chrome developer tips for minidump file debugging.

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.

Slow Name Resolution and nscd

For better scalability on nodes hosting many replicas, we recommend that you use `nscd` (name service cache daemon) to cache both DNS name resolution and static name resolution (via `/etc/hosts`).

When DNS lookups are slow, you will see a log message similar to the following:

```
W0926 11:19:01.339553 27231 net_util.cc:129] Time spent resolving address for
kudu-tserver.example.com: real 4.647s user 0.000s sys 0.000s
```

`nscd` can alleviate slow name resolution by providing a cache for the most common name service requests, such as for passwords, groups, and hosts.

Refer to your operating system documentation for how to install and enable `nscd`.

Consult your operating system's documentation for how to install and enable `nscd`.

Usability Issues

ClassNotFoundException: com.cloudera.kudu.hive.KuduStorageHandler

You will encounter this exception when you try to access a Kudu table using Hive. This is not a case of a missing jar, but simply that Impala stores Kudu metadata in Hive in a format that is unreadable to other tools, including Hive itself. and Spark. Currently, there is no workaround for Hive users. Spark users can work around this by creating temporary tables.

Runtime error: Could not create thread: Resource temporarily unavailable (error 11)

You will encounter this error when Kudu is unable to create more threads, usually on versions older than CDH 5.15 / Kudu 1.7. It happens on tablet servers, and is a sign that the tablet server hosts too many tablet replicas.

To fix the issue, you can raise the `nproc` ulimit as detailed in the documentation for your operating system or distribution.

However, the better solution is to reduce the number of replicas on the tablet server. This may involve rethinking the table's partitioning schema. For the recommended limits on number of replicas per tablet server, see the known issues and scaling limitations documentation.

Tombstoned or STOPPED tablet replicas

You may notice some replicas on a tablet server are in a STOPPED state and remain on the server indefinitely. These replicas are tombstones. A tombstone indicates that the tablet server once held a bona fide replica of its tablet. For example, in case a tablet server goes down and its replicas are re-replicated elsewhere, if the tablet server rejoins the cluster, its replicas will become tombstones. A tombstone will remain until the table it belongs to is deleted, or a new replica of the same tablet is placed on the tablet server. A count of tombstoned replicas and details of each one are available on the `/tablets` page of the tablet server web UI. The Raft consensus algorithm that Kudu uses for replication requires tombstones for correctness in certain rare situations. They consume minimal resources and hold no data. They must not be deleted.

Corruption: checksum error on CFile block

If the data on disk becomes corrupt, you will encounter warnings containing "Corruption: checksum error on CFile block" in the tablet server logs and client side errors when trying to scan tablets with corrupt CFile blocks. Fixing this corruption is a manual process.

To fix the issue, first identify all the affected tablets by running a checksum scan on the affected tables or tablets using the ksck tool.

```
sudo -u kudu kudu cluster ksck <master_addresses> -checksum_scan -tables=<tables>
sudo -u kudu kudu cluster ksck <master_addresses> -checksum_scan -tablets=<tablets>
```

If there is at least one replica for each tablet that does not return a corruption error, you can repair the bad copies by deleting them and forcing them to be re-replicated from the leader using the remote_replica delete tool.

```
sudo -u kudu kudu remote_replica delete <tserver_address> <tablet_id> "Cfile Corruption"
```

If all of the replica are corrupt, then some data loss has occurred. Until KUDU-2526 is completed, this can happen if the corrupt replica became the leader and the existing follower replicas are replaced.

If data has been lost, you can repair the table by replacing the corrupt tablet with an empty one using the unsafe_replace_tablet tool.

```
sudo -u kudu kudu tablet unsafe_replace_tablet <master_addresses> <tablet_id>
```

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.

Appendix: Apache License, Version 2.0

SPDX short identifier: Apache-2.0

Apache License
Version 2.0, January 2004
http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License.

Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License.

Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims

licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution.

You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

1. You must give any other recipients of the Work or Derivative Works a copy of this License; and
2. You must cause any modified files to carry prominent notices stating that You changed the files; and
3. You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
4. If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions.

Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks.

This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty.

Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability.

In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability.

While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

```
Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

  http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```