

cloudera[®]

Apache Kudu User Guide

Important Notice

© 2010-2017 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.

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.

1001 Page Mill Road, Bldg 3

Palo Alto, CA 94304

info@cloudera.com

US: 1-888-789-1488

Intl: 1-650-362-0488

www.cloudera.com

Release Information

Version: Apache Kudu 1.4.0 / CDH 5.12.x

Date: December 11, 2017

Table of Contents

About Apache Kudu.....	9
Concepts and Terms.....	10
<i>Columnar Datastore</i>	10
<i>Raft Consensus Algorithm</i>	10
<i>Table</i>	10
<i>Tablet</i>	10
<i>Tablet Server</i>	10
<i>Master</i>	10
<i>Catalog Table</i>	11
<i>Logical Replication</i>	11
Architectural Overview.....	11
Example Use Cases.....	12
Next Steps.....	13
Apache Kudu Release Notes.....	14
Schema Design and Usage Limitations.....	14
Kudu 1.4.0 / CDH 5.12.2 Release Notes.....	14
Kudu 1.4.0 / CDH 5.12.1 Release Notes.....	14
Kudu 1.4.0 / CDH 5.12.0 Release Notes.....	15
<i>New Features in Kudu 1.4.0 / CDH 5.12.0</i>	15
<i>Optimizations and Improvements in Kudu 1.4.0 / CDH 5.12.0</i>	15
<i>Fixed Issues in Kudu 1.4.0 / CDH 5.12.0</i>	16
<i>Wire Protocol Compatibility in Kudu 1.4.0 / CDH 5.12.0</i>	16
<i>Incompatible Changes in Kudu 1.4.0 / CDH 5.12.0</i>	16
<i>Known Issues and Limitations in Kudu 1.4.0 / CDH 5.12.0</i>	17
Kudu 1.3.0 / CDH 5.11.2 Release Notes.....	17
Kudu 1.3.0 / CDH 5.11.1 Release Notes.....	17
Kudu 1.3.0 / CDH 5.11.0 Release Notes.....	18
<i>New Features in Kudu 1.3.0 / CDH 5.11.0</i>	18
<i>Optimizations and Improvements in Kudu 1.3.0 / CDH 5.11.0</i>	18
<i>Fixed Issues in Kudu 1.3.0 / CDH 5.11.0</i>	19
<i>Wire Protocol Compatibility</i>	19
<i>Incompatible Changes in Kudu 1.3.0 / CDH 5.11.0</i>	20
<i>Known Issues and Limitations in Kudu 1.3.0 / CDH 5.11.0</i>	20
Kudu 1.2.0 / CDH 5.10.2 Release Notes.....	21
Kudu 1.2.0 / CDH 5.10.1 Release Notes.....	21

- Kudu 1.2.0 / CDH 5.10.0 Release Notes.....21
- New Features and Improvements in Kudu 1.2.0 / CDH 5.10.0.....21*
- Issues Fixed in Kudu 1.2.0 / CDH 5.10.0.....22*
- Incompatible Changes in Kudu 1.2.0 / CDH 5.10.0.....23*
- Known Issues and Limitations in Kudu 1.2.0 / CDH 5.10.0.....23*
- Kudu 1.1.x Release Notes.....23
- New Features in Kudu 1.1.0.....23*
- Issues Fixed in Kudu 1.1.0.....25*
- Kudu 1.0.1 Release Notes.....25
- Issues Fixed in Kudu 1.0.1.....25*
- Kudu 1.0.0 Release Notes.....25
- New Features in Kudu 1.0.0.....25*
- Incompatible Changes in Kudu 1.0.0.....26*
- Known Issues and Limitations of Kudu 1.0.0.....27*
- Issues Fixed in Kudu 1.0.0.....29*
- Kudu 0.10.0 Release Notes.....29
- New Features in Kudu 0.10.0.....29*
- Other Improvements in Kudu 0.10.0.....30*
- Issues Fixed in Kudu 0.10.0.....30*
- Incompatible Changes in Kudu 0.10.0.....30*
- Kudu 0.9.1 Release Notes.....31
- Issues Fixed in Kudu 0.9.1.....31*
- Kudu 0.9.0 Release Notes.....31
- New Features in Kudu 0.9.0.....31*
- Other Improvements and Changes in Kudu 0.9.0.....31*
- Issues Fixed in Kudu 0.9.0.....32*
- Incompatible Changes in Kudu 0.9.0.....32*
- Limitations of Kudu 0.9.0.....32*
- Upgrade Notes for Kudu 0.9.0.....32*
- Kudu 0.8.0 Release Notes.....32
- New Features in Kudu 0.8.0.....32*
- Other Improvements in Kudu 0.8.0.....33*
- Issues Fixed in Kudu 0.8.0.....33*
- Incompatible Changes in Kudu 0.8.0.....33*
- Limitations of Kudu 0.8.0.....33*
- Upgrade Notes for Kudu 0.8.0.....33*
- Kudu 0.7.1 Release Notes.....33
- Issues Fixed in Kudu 0.7.1.....33*
- Limitations of Kudu 0.7.1.....34*
- Upgrade Notes For Kudu 0.7.1.....34*
- Kudu 0.7.0 Release Notes.....34
- New Features in Kudu 0.7.0.....34*
- Other Improvements in Kudu 0.7.0.....34*
- Issues Fixed in Kudu 0.7.0.....34*

- Incompatible Changes in Kudu 0.7.0*.....34
- Limitations of Kudu 0.7.0*.....35
- Upgrade Notes For Kudu 0.7.0*.....36
- Kudu 0.6 Release Notes*.....36
- New Features in Kudu 0.6*.....37
- Issues Fixed in Kudu 0.6*.....37
- Limitations of Kudu 0.6*.....37
- Upgrade Notes For Kudu 0.6*.....38
- Kudu 0.5 Release Notes*.....39
- Limitations of Kudu 0.5*.....39
- Next Steps*.....40
- Apache Kudu Schema Design and Usage Limitations*.....40
- Schema Design Limitations*.....41
- Partitioning Limitations*.....42
- Scaling Recommendations and Limitations*.....42
- Server Management Limitations*.....42
- Cluster Management Limitations*.....42
- Replication and Backup Limitations*.....42
- Impala Integration Limitations*.....43
- Spark Integration Limitations*.....43
- Security Limitations*.....43
- Known Issues*.....44

Installing and Upgrading Apache Kudu.....46

- Kudu Installation Requirements*.....46
- Install Kudu Using Cloudera Manager*.....47
- Install Kudu Using Parcels*.....47
- Install Kudu Using Packages*.....48
- Install Kudu Using the Command Line*.....49
- Verify the Installation*.....51
- Upgrade Kudu using Cloudera Manager*.....52
- Upgrade Kudu Using Parcels*.....52
- Upgrade Kudu Using Packages*.....52
- Upgrade Kudu Using the Command Line*.....53
- Next Steps*.....54

Apache Kudu Configuration.....55

- Configuring the Kudu Master*.....55
- Configuring Tablet Servers*.....56

Apache Kudu Administration.....57

Starting and Stopping Kudu Processes.....	57
Kudu Web Interfaces.....	57
<i>Kudu Master Web Interface</i>	57
<i>Kudu Tablet Server Web Interface</i>	57
<i>Common Web Interface Pages</i>	57
Kudu Metrics.....	58
<i>Listing available metrics</i>	58
<i>Collecting metrics via HTTP</i>	58
<i>Collecting metrics to a log</i>	59
Common Kudu workflows.....	59
<i>Migrating to Multiple Kudu Masters</i>	60
<i>Recovering from a dead Kudu Master in a Multi-Master Deployment</i>	63
<i>Monitoring Cluster Health with ksck</i>	65
<i>Recovering from Disk Failure</i>	66
<i>Scaling Storage on Kudu Master and Tablet Servers in the Cloud</i>	67
Developing Applications With Apache Kudu.....	68
Viewing the API Documentation.....	68
Building the Java Client.....	68
Kudu Example Applications.....	69
Maven Artifacts.....	69
Kudu Python Client.....	69
Example Apache Impala Commands With Kudu.....	70
Kudu Integration with Spark.....	70
Integration with MapReduce, YARN, and Other Frameworks.....	72
Using Apache Impala with Kudu.....	73
Impala Database Containment Model.....	73
Internal and External Impala Tables.....	74
Using Impala To Query Kudu Tables.....	74
<i>Querying an Existing Kudu Table from Impala</i>	74
<i>Creating a New Kudu Table From Impala</i>	75
<i>Partitioning Tables</i>	75
<i>Optimizing Performance for Evaluating SQL Predicates</i>	79
<i>Inserting a Row</i>	79
<i>Updating a Row</i>	80
<i>Upserting a Row</i>	80
<i>Deleting a Row</i>	81
<i>Failures During INSERT, UPDATE, UPSERT, and DELETE Operations</i>	82
<i>Altering Table Properties</i>	82
<i>Dropping a Kudu Table using Impala</i>	82
Security Considerations.....	83

Known Issues and Limitations.....	83
Next Steps.....	83

Apache Kudu Security.....84

Kudu Authentication with Kerberos.....	84
<i>Internal Private Key Infrastructure (PKI)</i>	84
<i>Authentication Tokens</i>	84
<i>Scalability</i>	85
Encryption.....	85
Coarse-grained Authorization.....	85
Web UI Encryption.....	85
Web UI Redaction.....	86
Log Redaction.....	86
Configuring a Secure Kudu Cluster using Cloudera Manager.....	86
Configuring a Secure Kudu Cluster using the Command Line.....	88

Apache Kudu Schema Design.....89

The Perfect Schema.....	89
Column Design.....	89
<i>Column Encoding</i>	90
<i>Column Compression</i>	90
Primary Key Design.....	91
<i>Primary Key Index</i>	91
Partitioning.....	91
<i>Range Partitioning</i>	91
<i>Hash Partitioning</i>	92
<i>Multilevel Partitioning</i>	92
<i>Partition Pruning</i>	92
<i>Partitioning Examples</i>	92
Schema Alterations.....	95
Schema Design Limitations.....	95

Apache Kudu Transaction Semantics.....96

Single Tablet Write Operations.....	96
Writing to Multiple Tablets.....	96
Read Operations (Scans).....	97
Known Issues and Limitations.....	98
<i>Reads (Scans)</i>	98
<i>Writes</i>	99

Apache Kudu Background Maintenance Tasks.....100

Troubleshooting Apache Kudu.....102

- Issues Starting or Restarting the Master or Tablet Server.....102
- Error during hole punch test.....102*
- NTP Clock Synchronization Issues.....102*
- Usability Issues.....103
- Reporting Kudu Crashes Using Breakpad.....104
- Troubleshooting Performance Issues.....104
- Kudu Tracing.....104*

More Resources for Apache Kudu.....107

About Apache Kudu



Important: This is the documentation for Apache Kudu 1.4.0 / CDH 5.12.x. Documentation for Apache Kudu 1.5.0 / CDH 5.13.x (and higher) is available as part of the [Cloudera Enterprise documentation](#).

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

For more details, see [Example Use Cases](#) on page 12.

Kudu-Impala Integration Features

- **CREATE/ALTER/DROP TABLE** - Impala supports creating, altering, and dropping tables using Kudu as the persistence layer. The tables follow the same internal/external approach as other tables in Impala, allowing for flexible data ingestion and querying.
- **INSERT** - Data can be inserted into Kudu tables from Impala using the same mechanisms as any other table with HDFS or HBase persistence.
- **UPDATE/DELETE** - Impala supports the `UPDATE` and `DELETE` SQL commands to modify existing data in a Kudu table row-by-row or as a batch. The syntax of the SQL commands is 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.

Concepts and Terms

Columnar Datastore

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

Kudu network architecture

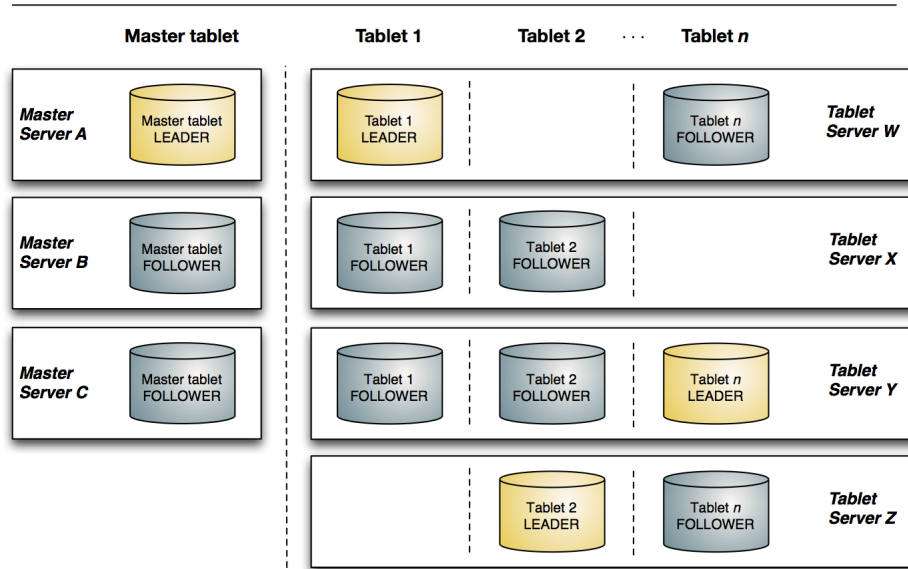


Figure 1: Kudu Architectural Overview

Example Use Cases

Streaming Input with Near Real Time Availability

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

Time-Series Application with Widely Varying Access Patterns

A time-series schema is one in which data points are organized and keyed according to the time at which they occurred. This can be useful for investigating the performance of metrics over time or attempting to predict future behavior based on past data. For instance, time-series customer data might be used both to store purchase click-stream history and to predict future purchases, or for use by a customer support representative. While these different types of analysis are occurring, inserts and mutations 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.

Next Steps

- Read about [installing Kudu](#).
- Learn about [using Kudu with Impala](#).

Apache Kudu Release Notes



Important: This is the documentation for Apache Kudu 1.4.0 / CDH 5.12.x. Documentation for Apache Kudu 1.5.0 / CDH 5.13.x (and higher) is available as part of the [Cloudera Enterprise documentation](#).

This topic includes the release notes for beta and generally available versions (GA) of Apache Kudu 1.4 (and lower). Release Notes for Apache Kudu 1.5 (and higher) are available as part of the [CDH release notes](#).

Schema Design and Usage Limitations

The [Apache Kudu Schema Design and Usage Limitations](#) on page 40 topic describes known issues and limitations with respect to schema design, integration with Apache Impala, and security in Kudu, as of the current release.

Kudu 1.4.0 / CDH 5.12.2 Release Notes

Apache Kudu 1.4.0 / CDH 5.12.2 is a bug-fix release which includes the following fixes:

- [KUDU-2209](#) - Fixed Kudu's handling of the STA_NANO flag, which caused spurious crashes due to one Kudu node thinking another node had a timestamp from the future.
- [KUDU-1788](#) - Increased the Raft RPC timeout to 30 seconds to avoid unsuccessful retries.
- [KUDU-2170](#) - Disallowed specifying the same address multiple times in Kudu's master list.
- [KUDU-2083](#) - Fixed an issue where, in rare circumstances, Kudu reduced the amount of maintenance ops it could run concurrently. The issue could cause the Kudu process to run out of memory and crash.
- [KUDU-1942](#) - Now Kerberos can successfully log in on hostnames with capital letters.
- [KUDU-2173](#) - Fixed an issue where scans with a predicate on a prefix of a range partition key could fail to return matching values.
- [KUDU-2167](#) - Fixed an issue where the C++ client would crash when the server filtered out all data in a scan RPC.
- [KUDU-2032](#) - Kerberos authentication no longer fails when `rdns = false` is configured in `krb5.conf`.
- Fixed a segmentation fault that could occur when running the `ksck` command against a master soon after the master was started.

Kudu 1.4.0 / CDH 5.12.1 Release Notes

Apache Kudu 1.4.0 / CDH 5.12.1 is a bug-fix release which includes the following fixes:

- [KUDU-2085](#) - Fixed a bug that caused crashes when seeking past the end of prefix-encoded blocks.
- [KUDU-2087](#) - Fixed an issue where Kudu would fail to start when Kerberos was enabled in FreeIPA-configured deployments.
- [KUDU-2053](#) - Fixed a race condition in the Java `RequestTracker`.
- [KUDU-2049](#) - Fixed an issue where scans on RLE-encoded integer columns would sometimes cause CHECK failures due to the CHECK condition being too strict.
- [KUDU-2052](#) - Kudu now uses `XFS_IOC_UNRESVSP64 ioctl` to punch holes on xfs filesystems. This fixes an issue with slow startup times on xfs when hole punching was done via `fallocate()`.
- [KUDU-1956](#) - Kudu will no longer crash if faced with a race condition when selecting rowsets for compaction.

Kudu 1.4.0 / CDH 5.12.0 Release Notes

Apache Kudu 1.4.0 / CDH 5.12.0 release is the first Kudu release to be supported with Cloudera Manager 5.12.x and CDH 5.12.x. Any future Kudu 1.4.x maintenance releases will be supported by corresponding CDH and Cloudera Manager 5.12.x maintenance releases.

New Features in Kudu 1.4.0 / CDH 5.12.0

- The C++ and Java client libraries now support the ability to alter the storage attributes (for example, encoding and compression) and default value of existing columns. Additionally, it is now possible to rename a column which is part of a table's primary key.
- The C++ client library now includes an experimental `KuduPartitioner` API which can be used to efficiently map rows to their associated partitions and hosts. This can be used to achieve better locality or distribution of writes in client applications.
- The Java client library now supports enabling fault tolerance on scanners. Fault tolerant scanners are able to transparently recover from concurrent server crashes at the cost of some performance overhead. See the [Java API documentation](#) for more details on usage.
- Kudu 1.4.0 includes the optional ability to compute, store, and verify checksums on all pieces of data stored on a server. Prior versions only performed checksums on certain portions of the stored data. This feature is not enabled by default since it makes a backward-incompatible change to the on-disk formats and thus prevents downgrades.
- **Kudu Command Line Tools**
 - The `kudu` command line tool now includes a new advanced administrative command, `kudu remote_replica unsafe_change_config`. This command can be used to force a tablet to perform an unsafe change of its Raft replication configuration. This can then be used to recover from scenarios such as a loss of a majority of replicas, at the risk of losing edits.
 - A new `kudu fs check` command can be used to perform various offline consistency checks on the local on-disk storage of a Kudu Tablet Server or Master. In addition to detecting various inconsistencies or corruptions, it can also detect and remove data blocks that are no longer referenced by any tablet but were not fully removed from disk due to a crash or a bug in prior versions of Kudu.
 - The `kudu` command line tool can now be used to list the addresses and identifiers of the servers in the cluster using either `kudu master list` or `kudu tserver list`.

For a complete list of Kudu's command line tools and usage information, see the [Command Line Reference documentation](#).

Optimizations and Improvements in Kudu 1.4.0 / CDH 5.12.0

- `kudu cluster kscck` now detects and reports new classes of inconsistencies and issues. In particular, it is better able to detect cases where a configuration change such as a replica eviction or addition is pending but is unable to be committed. It also properly detects and reports cases where a tablet has no elected leader.
- The default size for Write Ahead Log (WAL) segments has been reduced from 64MB to 8MB. Additionally, in the case that all replicas of a tablet are fully up to date and data has been flushed from memory, servers will now retain only a single WAL segment rather than two. These changes are expected to reduce the average consumption of disk space on the configured WAL disk by 16x, as well as improve the startup speed of tablet servers by reducing the number and size of WAL segments that need to be re-read.
- The default on-disk storage system used by Kudu servers (Log Block Manager) has been improved to compact its metadata and remove dead containers. This compaction and garbage collection occurs only at startup. Thus, the first startup after upgrade is expected to be longer than usual, and subsequent restarts should be shorter.
- The usability of the Kudu web interfaces has been improved, particularly for cases where a server hosts many tablets or a table has many partitions. Pages that list tablets now include a top-level summary of tablet status and show the complete list under a toggleable section.
- 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. This can improve the throughput of write-heavy workloads.

- 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.
- Many performance improvements were made to improve write performance. Applications which send large batches of writes to Kudu should see substantially improved throughput in Kudu 1.4.0.
- Several improvements were made to reduce the memory consumption of Kudu Tablet Servers which hold large volumes of data. The specific amount of memory saved varies depending on workload, but the expectation is that approximately 350MB of excess peak memory usage has been eliminated per TB of data stored.
- The number of threads used by the Kudu Tablet Server has been reduced. Previously, each tablet used a dedicated thread to append to its WAL. Those threads now automatically stop running if there is no activity on a given tablet for a short period of time.

Fixed Issues in Kudu 1.4.0 / CDH 5.12.0

- [KUDU-2020](#) - Fixed an issue where re-replication after a failure would proceed significantly slower than expected. This bug caused many tablets to be unnecessarily copied multiple times before successfully being considered re-replicated, resulting in significantly more network and IO bandwidth usage than expected. Mean time to recovery on clusters with large amounts of data is improved by up to 10x by this fix.
- [KUDU-1982](#) - Fixed an issue where the Java client would call `NetworkInterface.getByInetAddress` very often, causing performance problems particularly on Windows where this function can be quite slow.
- [KUDU-1755](#) - Improved the accuracy of the `on_disk_size` replica metrics to include the size consumed by bloom filters, primary key indexes, superbloc metadata, and delta files. Note that because the size metric is now more accurate, the reported values are expected to increase after upgrading to Kudu 1.4.0. This does not indicate that replicas are using more space after the upgrade; rather, it is now accurately reporting the amount of space that has always been used.
- [KUDU-1192](#) - Kudu servers will now periodically flush their log messages to disk even if no `WARNING`-level messages have been logged. This makes it easier to tail the logs to see progress output during normal startup.

Wire Protocol Compatibility in Kudu 1.4.0 / CDH 5.12.0

- Kudu 1.4.0 clients can connect to servers running Kudu 1.0.0 or later. If the client uses features that are not available on the target server, an error will be returned.
- Kudu 1.0.0 clients can connect to servers running Kudu 1.4.0 with the exception of the restrictions on secure clusters mentioned further on in this list.
- Rolling upgrade between Kudu 1.3.0 and Kudu 1.4.0 servers is believed to be possible though it has not been sufficiently tested. Users are encouraged to shut down all nodes in the cluster, upgrade the software, and then restart the daemons on the new version.
- The authentication features introduced in Kudu 1.3.0 place the following limitations on wire compatibility between Kudu 1.4.0 and versions earlier than 1.3.0:
 - If a Kudu 1.4.0 cluster is configured with authentication or encryption set to `required`, clients older than Kudu 1.3.0 will be unable to connect.
 - If a Kudu 1.4.0 cluster is configured with authentication and encryption set to `optional` or `disabled`, older clients will still be able to connect.

Incompatible Changes in Kudu 1.4.0 / CDH 5.12.0

Kudu servers, by default, will now only allow unencrypted or unauthenticated connections from trusted subnets, which are private networks (127.0.0.0/8, 10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16, 169.254.0.0/16) and local subnets of all local network interfaces. Unencrypted or unauthenticated connections from publicly routable IPs will be rejected, even if encryption and authentication are not configured.

The trusted subnets can be configured using the `--trusted_subnets` flag, which can be set to IP blocks represented in CIDR notation separated by comma. Set it to '0.0.0.0/0' to allow unauthenticated connections from all remote IP addresses. However, if network access is not otherwise restricted by a firewall, malicious users may be able to gain unauthorized access. This can be mitigated if authentication and encryption are configured to be required.

Client Library Compatibility

- **Java** - The Kudu 1.4.0 Java client library is API- and ABI-compatible with Kudu 1.3.0. Applications written against Kudu 1.3.0 will compile and run against the Kudu 1.4.0 client library and vice-versa, unless one of the following newly-added APIs is used:
 - `[Async]KuduScannerBuilder.setFaultTolerant(...)`
 - **New methods in `AlterTableOptions`:** `removeDefault`, `changeDefault`, `changeDesiredBlockSize`, `changeEncoding`, `changeCompressionAlgorithm`
 - `KuduClient.updateLastPropagatedTimestamp`
 - `KuduClient.getLastPropagatedTimestamp`
 - **New getters in `PartialRow`:** `getBoolean`, `getByte`, `getShort`, `getInt`, `getLong`, `getFloat`, `getDouble`, `getString`, `getBinaryCopy`, `getBinary`, `isNull`, `isSet`.
- **C++** - The Kudu 1.4.0 C++ client is API- and ABI-forward-compatible with Kudu 1.3.0. Applications written and compiled against the Kudu 1.3.0 client library will run without modification against the Kudu 1.4.0 client library. Applications written and compiled against the Kudu 1.4.0 client library will run without modification against the Kudu 1.3.0 client library unless they use one of the following new APIs:
 - `KuduPartitionerBuilder`
 - `KuduPartitioner`
 - `KuduScanner::SetRowFormatFlags` (unstable API)
 - `KuduScanBatch::direct_data`, `KuduScanBatch::indirect_data` (unstable API)
- **Python** - The Kudu 1.4.0 Python client is API-compatible with Kudu 1.3.0. Applications written against Kudu 1.3.0 will continue to run against the Kudu 1.4.0 client and vice-versa.

Known Issues and Limitations in Kudu 1.4.0 / CDH 5.12.0

For a complete list of schema design and usage limitations for Apache Kudu, see [Apache Kudu Schema Design and Usage Limitations](#) on page 40.

Kudu 1.3.0 / CDH 5.11.2 Release Notes

Apache Kudu 1.3.0 / CDH 5.11.2 is a bug-fix release which includes the following fixes:

- [KUDU-2053](#) - Fixed a race condition in the Java `RequestTracker`.
- [KUDU-2049](#) - Fixed an issue where scans on RLE-encoded integer columns would sometimes cause CHECK failures due to the CHECK condition being too strict.
- [KUDU-1963](#) - Fixed an issue where the Java client misdiagnoses an error and logs a `NullPointerException` when a connection is closed by client while a negotiation is in progress.
- [KUDU-1853](#) - Fixed an issue where data blocks could be orphaned after a failed tablet copy.

Kudu 1.3.0 / CDH 5.11.1 Release Notes

Apache Kudu 1.3.0 / CDH 5.11.1 is a bug-fix release which includes the following fixes:

- [KUDU-1999](#) - Fixed an issue where the Kudu Spark connector would fail to kinit with the principal and keytab provided to a job.
- [KUDU-1993](#) - Fixed a validation issue with grouped gflags.

- [KUDU-1981](#) - Fixed an issue where Kudu server components would fail to start on machines with fully-qualified domain names longer than 64 characters when security was enabled. This was due to hard-coded restrictions in the OpenSSL library.
- [KUDU-1607](#) - Fixed a case in which a tablet replica on a tablet server could retain blocks of data which prevented it from being fully deleted.
- [KUDU-1933](#) - Fixed an issue in which a tablet server would crash and fail to restart after a single tablet received more than two billion write operations.
- [KUDU-1964](#) - Fixed a performance degradation issue caused by OpenSSL locks under high concurrency.

Kudu 1.3.0 / CDH 5.11.0 Release Notes



Note: Apache Kudu 1.3.0 / CDH 5.11.0 release is the first Kudu release to be supported with Cloudera Manager 5.11.x and CDH 5.11.x. Any future Kudu 1.3.x maintenance releases will be supported by corresponding CDH and Cloudera Manager 5.11.x maintenance releases.

New Features in Kudu 1.3.0 / CDH 5.11.0

- The Kudu CSD is now included in Cloudera Manager. Manual installation of the Kudu CSD is not required.
- Kudu 1.3 supports strong authentication using Kerberos. This feature allows users to authenticate themselves to Kudu using Kerberos tickets, and also provides mutual authentication of servers using Kerberos credentials stored in keytabs.
- Added support for encryption of data on the network using Transport Layer Security (TLS). Kudu will now use TLS to encrypt all network traffic between clients and servers as well as any internal traffic among servers, with the exception of traffic determined to be within a localhost network connection. Encryption is enabled by default whenever it can be determined that both the client and server support the feature.
- Added support for coarse-grained service-level authorization checks for access to the cluster. The operator may set up lists of permitted users who may act as administrators and as clients of the cluster. Combined with the strong authentication feature described above, this can enable a secure environment for some use cases. Note that fine-grained access control, for example table-level or column-level, is not yet supported.
- A background task was added to tablet servers that removes historical versions of data that have fallen behind the configured data retention time. This reduces disk space usage in all workloads, but particularly in those with a higher volume of updates or upserts. After upgrading, you might see disk usage go down a lot.
- Kudu now incorporates Google Breakpad, a library that writes crash reports in the case of a server crash. These reports can be found by default under the `minidumps` directory, or within a configured log directory, and can be useful during bug diagnosis.
- As of Kudu 1.3, multi-word flags can use a dash '-' separator in lieu of the underscore '_' separator. For example, the `--memory_limit_hard_bytes` flag can now be specified as `--memory-limit-hard-bytes`, or even `--memory_limit-hard_bytes`.

Optimizations and Improvements in Kudu 1.3.0 / CDH 5.11.0

- Kudu servers will now change the file permissions of data directories and contained data files based on a new configuration flag `--umask`. As a result, after upgrading, permissions on disk may be more restrictive than in previous versions. The new default configuration improves data security.
- Kudu web UIs will now redact strings that might include sensitive user data. For example, the monitoring page, which shows in-progress scans, no longer includes the scanner predicate values. The tracing and RPC diagnostics endpoints no longer include contents of RPCs, which may include table data.
- By default, Kudu now reserves 1% of each configured data volume as free space. If a volume is seen to have less than 1% of disk space free, Kudu stops writing to that volume to avoid completely filling up the disk.

- The default encoding for numeric columns (`int`, `float`, and `double`) has been changed to `BIT_SHUFFLE`. The default encoding for `binary` and `string` columns has been changed to `DICTIONARY_ENCODING`. Dictionary encoding automatically falls back to the old default (`PLAIN`) when cardinality is too high to be effectively encoded. These new defaults match the default behavior of other storage mechanisms, such as Apache Parquet, and are likely to perform better out of the box.

Existing tables with `AUTO_ENCODING`-configured columns will only be encoded during `DiskRowSet` compactions, or if new data is written to them. New tables and columns will have the new defaults.

- Kudu now uses `LZ4` compression when writing its Write Ahead Log (WAL). This improves write performance and stability for many use cases.
- Kudu now uses `LZ4` compression when writing delta files. This can improve both read and write performance as well as save substantial disk usage, especially for workloads involving a high number of updates or upserts containing compressible data.

Existing delta files will not get compressed until they are compacted. New files on both old and new tables alike will be compressed.

- The Kudu API now supports the ability to express `IS NULL` and `IS NOT NULL` predicates on scanners. The Spark DataSource integration will take advantage of these new predicates when possible.
- Both C++ and Java clients have been optimized to prune partitions more effectively when performing scans using the `IN (...)` predicate.
- The exception messages produced by the Java client are now truncated to a maximum length of 32KB.

Fixed Issues in Kudu 1.3.0 / CDH 5.11.0

- [KUDU-1968](#) - Fixed an issue in which the tablet server would delete an incorrect set of data blocks after an aborted attempt to copy a tablet from another server. This would produce data loss in unrelated tablets.
- [KUDU-1962](#) - Fixed a `NullPointerException` in the Java client in the case that the Kudu master is overloaded at the time the client requests location information. This could cause client applications to hang indefinitely regardless of configured timeouts.
- [KUDU-1893](#) - Fixed a critical bug in which wrong results would be returned when evaluating predicates applied to columns added using the `ALTER TABLE` operation.
- [KUDU-1905](#) - Fixed an issue where Kudu would crash after reinserts that resulted in an empty change list. This occurred in cases where the primary key was composed of all columns.
- [KUDU-1899](#) - Fixed a crash that occurred after inserting a row with an empty string as the single-column primary key.
- [KUDU-1904](#) - Fixed a potential crash when performing random reads against a column using RLE encoding and containing long runs of `NULL` values.
- [KUDU-1856](#) - Fixed an issue in which disk space could be leaked by Kudu servers storing data on partitions using the XFS file system. Any leaked disk space will now be automatically recovered upon upgrade.
- [KUDU-1888](#), [KUDU-1906](#) - Fixed multiple issues in the Java client where operation callbacks would never be triggered, causing the client to hang.

Wire Protocol Compatibility

Kudu 1.3.0 is wire-compatible with previous versions of Kudu:

- Kudu 1.3.0 clients can connect to servers running Kudu 1.0.0. If the client uses features that are not available on the target server, an error will be returned.
- Kudu 1.0.0 clients can connect to servers running Kudu 1.3.0 with the exception of the below-mentioned restrictions regarding secure clusters.

- Rolling upgrade between Kudu 1.2.0 and Kudu 1.3.0 servers is believed to be possible though has not been sufficiently tested. Users are encouraged to shut down all nodes in the cluster, upgrade the software, and then restart the daemons on the new version.

Security - The authentication features introduced in Kudu 1.3.0 place the following limitations on wire compatibility with older versions:

- If a Kudu 1.3.0 cluster is configured with authentication or encryption set to `required`, older clients will not be able to connect.
- If a Kudu 1.3.0 cluster is configured with authentication and encryption set to `optional` or `disabled`, older clients will still be able to connect.

Incompatible Changes in Kudu 1.3.0 / CDH 5.11.0

- Due to storage format changes in Kudu 1.3, downgrading from Kudu 1.3 to earlier versions is not supported. After upgrading to Kudu 1.3, attempting to restart with an earlier version will result in an error.
- In order to support running MapReduce and Spark jobs on secure clusters, these frameworks now connect to the cluster at job submission time to retrieve authentication credentials which can later be used by the tasks to be spawned. This means that the process submitting jobs to Kudu clusters must have direct access to that cluster.
- The embedded web servers in Kudu processes now specify the `X-Frame-Options: DENY` HTTP header which prevents embedding of Kudu web pages in HTML `iframe` elements.

Client Library Compatibility

- **Java** - The Kudu 1.3.0 Java client library is API- and ABI-compatible with Kudu 1.2.0. Applications written against Kudu 1.2.0 will compile and run against the Kudu 1.3.0 client library and vice-versa, unless one of the following newly-added APIs is used:
 - `[Async]KuduClient.exportAuthenticationCredentials(...)` (unstable API)
 - `[Async]KuduClient.importAuthenticationCredentials(...)` (unstable API)
 - `[Async]KuduClient.getMasterAddressesAsString()`
 - `KuduPredicate.newIsNotNullPredicate()`
 - `KuduPredicate.newIsNullPredicate()`
- **C++** - The Kudu 1.3.0 C++ client is API- and ABI-forward-compatible with Kudu 1.2.0. Applications written and compiled against the Kudu 1.2.0 client library will run without modification against the Kudu 1.3.0 client library. Applications written and compiled against the Kudu 1.3.0 client library will run without modification against the Kudu 1.2.0 client library unless they use one of the following new APIs:
 - `kudu::DisableOpenSSLInitialization()`
 - `KuduClientBuilder::import_authentication_credentials(...)`
 - `KuduClient::ExportAuthenticationCredentials(...)`
 - `KuduClient::NewIsNotNullPredicate(...)`
 - `KuduClient::NewIsNullPredicate(...)`
- **Python** - The Kudu 1.3.0 Python client is API-compatible with Kudu 1.2.0. Applications written against Kudu 1.2.0 will continue to run against the Kudu 1.3.0 client and vice-versa.

Known Issues and Limitations in Kudu 1.3.0 / CDH 5.11.0

For a complete list of schema design and usage limitations for Apache Kudu, see [Apache Kudu Schema Design and Usage Limitations](#) on page 40.

Kudu 1.2.0 / CDH 5.10.2 Release Notes

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

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

Kudu 1.2.0 / CDH 5.10.1 Release Notes

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

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

Kudu 1.2.0 / CDH 5.10.0 Release Notes



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

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

New Features and Improvements in Kudu 1.2.0 / CDH 5.10.0

See also [Issues resolved for Kudu 1.2.0](#) and [Git changes between 1.1.x and 1.2.x](#).

New Features

- Kudu clients and servers now redact user data such as cell values from log messages, Java exception messages, and Status strings. User metadata such as table names, column names, and partition bounds are not redacted.
- Kudu's ability to provide consistency guarantees has been substantially improved:
 - Replicas now correctly track their "safe timestamp". This timestamp is the maximum timestamp at which reads are guaranteed to be repeatable.
 - A scan created using the `SCAN_AT_SNAPSHOT` mode will now either wait for the requested snapshot to be "safe" at the replica being scanned, or be re-routed to a replica where the requested snapshot is "safe". This ensures that all such scans are repeatable.
 - Kudu Tablet Servers now properly retain historical data when a row with a given primary key is inserted and deleted, followed by the insertion of a new row with the same key. Previous versions of Kudu would not retain history in such situations. This allows the server to return correct results for snapshot scans with a timestamp in the past, even in the presence of such "reinsertion" scenarios.
 - The Kudu clients now automatically retain the timestamp of their latest successful read or write operation. Scans using the `READ_AT_SNAPSHOT` mode without a client-provided timestamp automatically assign a timestamp higher than the timestamp of their most recent write. Writes also propagate the timestamp, ensuring that sequences of operations with causal dependencies between them are assigned increasing

timestamps. Together, these changes allow clients to achieve read-your-writes consistency, and also ensure that snapshot scans performed by other clients return causally-consistent results.

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

Optimizations and Improvements

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

Issues Fixed in Kudu 1.2.0 / CDH 5.10.0

See [Issues resolved for Kudu 1.2.0](#) and [Git changes between 1.1.x and 1.2.x](#).

- [KUDU-1508](#) - Fixed a long-standing issue in which running Kudu on `ext4` file systems could cause file system corruption. While this issue has been known to still manifest in certain rare cases, the corruption is harmless and can be repaired as part of a regular `fsck`. Switching from `ext4` to `xfs` will also solve the problem.
- [KUDU-1399](#) - Implemented an LRU cache for open files, which prevents running out of file descriptors on long-lived Kudu clusters. By default, Kudu will limit its file descriptor usage to half of its configured `ulimit`.
- [Gerrit #5192](#) - Fixed an issue which caused data corruption and crashes in the case that a table had a non-composite (single-column) primary key, and that column was specified to use `DICT_ENCODING` or `BITSHUFFLE` encodings. If a table with an affected schema was written in previous versions of Kudu, the corruption will not be automatically repaired; users are encouraged to re-insert such tables after upgrading to Kudu 1.2 or later.
- [Gerrit #5541](#) - Fixed a bug in the Spark `KuduRDD` implementation which could cause rows in the result set to be silently skipped in some cases.
- [KUDU-1551](#) - Fixed an issue in which the tablet server would crash on restart in the case that it had previously crashed during the process of allocating a new WAL segment.
- [KUDU-1764](#) - Fixed an issue where Kudu servers would leak approximately 16-32MB of disk space for every 10GB of data written to disk. After upgrading to Kudu 1.2 or later, any disk space leaked in previous versions will be automatically recovered on startup.
- [KUDU-1750](#) - Fixed an issue where the API to drop a range partition would drop any partition with a matching lower `_or_` upper bound, rather than any partition with matching lower `_and_` upper bound.
- [KUDU-1766](#) - Fixed an issue in the Java client where equality predicates which compared an integer column to its maximum possible value (e.g. `Integer.MAX_VALUE`) would return incorrect results.

- [KUDU-1780](#) - Fixed the kudu-client Java artifact to properly shade classes in the `com.google.thirdparty` namespace. The lack of proper shading in prior releases could cause conflicts with certain versions of Google Guava.
- [Gerrit #5327](#) - Fixed shading issues in the `kudu-flume-sink` Java artifact. The sink now expects that Hadoop dependencies are provided by Flume, and properly shades the Kudu client's dependencies.
- Fixed a few issues using the Python client library from Python 3.

Incompatible Changes in Kudu 1.2.0 / CDH 5.10.0

Apache Kudu 1.2.0 introduces the following incompatible changes:

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

Known Issues and Limitations in Kudu 1.2.0 / CDH 5.10.0

- [KUDU-1893](#) - Certain query results incorrectly include rows with null values for predicates.
Solution - Upgrade to the latest maintenance release.
- **Schema and Usage Limitations** - For a complete list of schema design and usage limitations for Apache Kudu, see [Apache Kudu Schema Design and Usage Limitations](#) on page 40.

Kudu 1.1.x Release Notes

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

New Features in Kudu 1.1.0

See also [Issues resolved for Kudu 1.1.0](#) and [Git changes between 1.0.x and 1.1.x](#).

- The Python client has been brought up to feature parity with the Java and C++ clients and as such the package version will be brought to 1.1 with this release (from 0.3). A list of the highlights can be found below.
 - Improved Partial Row semantics
 - Range partition support
 - Scan Token API
 - Enhanced predicate support
 - Support for all Kudu data types (including a mapping of Python's `datetime.datetime` to `UNIXTIME_MICROS`)

- Alter table support
 - Enabled Read at Snapshot for Scanners
 - Enabled Scanner Replica Selection
 - A few bug fixes for Python 3 in addition to various other improvements.
- `IN LIST` predicate pushdown support was added to allow optimized execution of filters which match on a set of column values. Support for Spark, Map Reduce and Impala queries utilizing `IN LIST` pushdown is not yet complete.
 - The Java client now features client-side request tracing in order to help troubleshoot timeouts. Error messages are now augmented with traces that show which servers were contacted before the timeout occurred instead of just the last error. The traces also contain RPCs that were required to fulfill the client's request, such as contacting the master to discover a tablet's location. Note that the traces are not available for successful requests and cannot be queried.

Performance

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

Wire protocol compatibility

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

Command line tools

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

Client APIs (C++/Java/Python)

- The C++ client no longer requires the [old gcc5 ABI](#). Which ABI is actually used depends on the compiler configuration. Some new distros (e.g. Ubuntu 16.04) will use the new ABI. Your application must use the same ABI as is used by the client library; an easy way to guarantee this is to use the same compiler to build both.
- The C++ client's `KuduSession::CountBufferedOperations()` method is deprecated. Its behavior is inconsistent unless the session runs in the `MANUAL_FLUSH` mode. Instead, to get number of buffered operations, count invocations of the `KuduSession::Apply()` method since last `KuduSession::Flush()` call or, if using asynchronous flushing, since last invocation of the callback passed into `KuduSession::FlushAsync()`.
- The Java client's `OperationResponse.getWriteTimestamp` method was renamed to `getWriteTimestampRaw` to emphasize that it doesn't return milliseconds, unlike what its Javadoc indicated. The renamed method was also hidden from the public APIs and should not be used.

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

Issues Fixed in Kudu 1.1.0

See [Issues resolved for Kudu 1.1.0](#) and [Git changes between 1.0.x and 1.1.x](#).

Kudu 1.0.1 Release Notes

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

Issues Fixed in Kudu 1.0.1

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

Kudu 1.0.0 Release Notes

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

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

To upgrade Kudu to 1.0.0, see [Upgrade Parcels](#) or [Upgrade Packages](#).

Other Noteworthy Changes

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

New Features in Kudu 1.0.0

See also [Issues resolved for Kudu 1.0.0](#) and [Git changes between 0.10.0 and 1.0.0](#).

- Removal of multiversion concurrency control (MVCC) history is now supported. This is known as tablet history GC. This allows Kudu to reclaim disk space, where previously Kudu would keep a full history of all changes made to a given table since the beginning of time. Previously, the only way to reclaim disk space was to drop a table.
- Kudu will still keep historical data, and the amount of history retained is controlled by setting the configuration flag `--tablet_history_max_age_sec`, which defaults to 15 minutes (expressed in seconds). The timestamp represented by the current time minus `tablet_history_max_age_sec` is known as the ancient history mark

(AHM). When a compaction or flush occurs, Kudu will remove the history of changes made prior to the ancient history mark. This only affects historical data; currently-visible data will not be removed. A specialized maintenance manager background task to remove existing “cold” historical data that is not in a row affected by the normal compaction process will be added in a future release.

- Most of Kudu’s command line tools have been consolidated under a new top-level `kudu` tool. This reduces the number of large binaries distributed with Kudu and also includes much-improved help output.
- The Kudu Flume Sink now supports processing events containing Avro-encoded records, using the new `AvroKuduOperationsProducer`.
- Administrative tools including `kudu cluster ksck` now support running against multi-master Kudu clusters.
- The output of the `ksck` tool is now colorized and much easier to read.
- The C++ client API now supports writing data in `AUTO_FLUSH_BACKGROUND` mode. This can provide higher throughput for ingest workloads.

Performance

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

Wire protocol compatibility

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

Incompatible Changes in Kudu 1.0.0

Command line tools

- The `kudu-pbc-dump` tool has been removed. The same functionality is now implemented as `kudu pbc dump`.
- The `kudu-ksck` tool has been removed. The same functionality is now implemented as `kudu cluster ksck`.
- The `cfile-dump` tool has been removed. The same functionality is now implemented as `kudu fs cfile dump`.
- The `log-dump` tool has been removed. The same functionality is now implemented as `kudu wal dump` and `kudu local_replica dump wals`.
- The `kudu-admin` tool has been removed. The same functionality is now implemented within `kudu table` and `kudu tablet`.
- The `kudu-fs_dump` tool has been removed. The same functionality is now implemented as `kudu fs dump`.

- The `kudu-ts-cli` tool has been removed. The same functionality is now implemented within `kudu master`, `kudu remote_replica`, and `kudu tserver`.
- The `kudu-fs_list` tool has been removed and some similar useful functionality has been moved under `kudu local_replica`.

Configuration flags

Some configuration flags are marked 'unsafe' and 'experimental'. Such flags are disabled by default. You 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](#).

Client APIs (C++/Java/Python)

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

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

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

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

Apache Flume Integration

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

Known Issues and Limitations of Kudu 1.0.0

Schema and Usage Limitations

- Kudu is primarily designed for analytic use cases. You are likely to encounter issues if a single row contains multiple kilobytes of data.
- The columns which make up the primary key must be listed first in the schema.
- Key columns cannot be altered. You must drop and recreate a table to change its keys.
- Key columns must not be null.
- Columns with `DOUBLE`, `FLOAT`, or `BOOL` types are not allowed as part of a primary key definition.
- Type and nullability of existing columns cannot be changed by altering the table.
- A table's primary key cannot be changed.
- Dropping a column does not immediately reclaim space. Compaction must run first. There is no way to run compaction manually, but dropping the table will reclaim the space immediately.

Partitioning Limitations

- Tables must be manually pre-split into tablets using simple or compound primary keys. Automatic splitting is not yet possible. Range partitions may be added or dropped after a table has been created. See Schema Design for more information.

- Data in existing tables cannot currently be automatically repartitioned. As a workaround, create a new table with the new partitioning and insert the contents of the old table.

Replication and Backup Limitations

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

Impala Limitations

- To use Kudu with Impala, you must install a special release of Impala called Impala_Kudu. Obtaining and installing a compatible Impala release is detailed in [Using Apache Impala with Kudu](#) on page 73.
- To use Impala_Kudu alongside an existing Impala instance, you must install using parcels.
- Updates, inserts, and deletes via Impala are non-transactional. If a query fails part of the way through, its partial effects will not be rolled back.
- All queries will be distributed across all Impala hosts which host a replica of the target table(s), even if a predicate on a primary key could correctly restrict the query to a single tablet. This limits the maximum concurrency of short queries made via Impala.
- No `TIMESTAMP` and `DECIMAL` type support. (The underlying Kudu type formerly known as `TIMESTAMP` has been renamed to `UNIXTIME_MICROS`; currently, there is no Impala-compatible `TIMESTAMP` type.)
- The maximum parallelism of a single query is limited to the number of tablets in a table. For good analytic performance, aim for 10 or more tablets per host or use large tables.
- Impala is only able to push down predicates involving `=`, `<=`, `>=`, or `BETWEEN` comparisons between any column and a literal value, and `<` and `>` for integer columns only. For example, for a table with an integer key `ts`, and a string key `name`, the predicate `WHERE ts >= 12345` will convert into an efficient range scan, whereas `where name > 'lipcon'` will currently fetch all data from the table and evaluate the predicate within Impala.

Security Limitations

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

Client and API Limitations

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

Other Known Issues

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

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

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

Issues Fixed in Kudu 1.0.0

See [Issues resolved for Kudu 1.0.0](#) and [Git changes between 0.10.0 and 1.0.0](#).

Kudu 0.10.0 Release Notes

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

See also [Issues resolved for Kudu 0.10.0](#) and [Git changes between 0.9.1 and 0.10.0](#).

To upgrade Kudu to 0.10.0, see [Upgrade Parcels](#) or [Upgrade Packages](#).

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

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

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

This release does not maintain full Java API or ABI compatibility with Kudu 0.9.x due to a package rename and some other small changes. See [Incompatible Changes in Kudu 0.10.0](#) on page 30 for details.

Other Noteworthy Changes

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

New Features in Kudu 0.10.0

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

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

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

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

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

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

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

Other Improvements in Kudu 0.10.0

- [KUDU-1516](#): The `kudu-ksck` tool has been improved and now detects problems such as when a tablet does not have a majority of replicas on live tablet servers, or if those replicas aren't in a good state. Users who currently depend on the tool to detect inconsistencies may now see failures when before they wouldn't see any.
- [Gerrit #3477](#): The way operations are buffered in the Java client has been reworked. Previously, the session's buffer size was set per tablet, meaning that a buffer size of 1,000 for 10 tablets being written to allowed for 10,000 operations to be buffered at the same time. With this change, all the tablets share one buffer, so users might need to set a bigger buffer size in order to reach the same level of performance as before.
- [Gerrit #3674](#): Added `LESS` and `GREATER` options for column predicates.
- [KUDU-1444](#): Added support for passing back basic per-scan metrics, such as cache hit rate, from the server to the C++ client. See the `KuduScanner::GetResourceMetrics()` API for detailed usage. This feature will be supported in the Java client API in a future release.
- [KUDU-1446](#): Improved the order in which the tablet server evaluates predicates, so that predicates on smaller columns are evaluated first. This may improve performance on queries which apply predicates on multiple columns of different sizes.
- [KUDU-1398](#): Improved the storage efficiency of Kudu's internal primary key indexes. This optimization should decrease space usage and improve random access performance, particularly for workloads with lengthy primary keys.

Issues Fixed in Kudu 0.10.0

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

Incompatible Changes in Kudu 0.10.0

- [Gerrit #3737](#): The Java client has been repackaged under `org.apache.kudu` instead of `org.kududb`. Import statements for Kudu classes must be modified in order to compile against 0.10.0. Wire compatibility is maintained.
- [Gerrit #3055](#): The Java client's synchronous API methods now throw `KuduException` instead of `Exception`. Existing code that catches `Exception` should still compile, but introspection of an exception's message may be impacted. This change was made to allow thrown exceptions to be queried more easily using `KuduException.getStatus` and calling one of `Status`'s methods. For example, an operation that tries to delete a table that doesn't exist would return a `Status` that returns true when queried on `isNotFound()`.
- The Java client's `KuduTable.getTabletsLocations` set of methods is now deprecated. Additionally, they now take an exclusive end partition key instead of an inclusive key. Applications are encouraged to use the scan tokens API instead of these methods in the future.
- The C++ API for specifying split points on range-partitioned tables has been improved to make it easier for callers to properly manage the ownership of the provided rows.

- The `TableCreator::split_rows` API took a `vector<const KuduPartialRow*>`, which made it very difficult for the calling application to do proper error handling with cleanup when setting the fields of the `KuduPartialRow`. This API has been now been deprecated and replaced by a new method `TableCreator::add_range_split` which allows easier use of smart pointers for safe memory management.
- The Java client's internal buffering has been reworked. Previously, the number of buffered write operations was constrained on a per-tablet-server basis. Now, the configured maximum buffer size constrains the total number of buffered operations across all tablet servers in the cluster. This provides a more consistent bound on the memory usage of the client regardless of the size of the cluster to which it is writing. This change can negatively affect the write performance of Java clients which rely on buffered writes. Consider using the `setMutationBufferSize` API to increase a session's maximum buffer size if write performance seems to be degraded after upgrading to Kudu 0.10.0.
- The "remote bootstrap" process used to copy a tablet replica from one host to another has been renamed to "Tablet Copy". This resulted in the renaming of several RPC metrics. Any users previously explicitly fetching or monitoring metrics related to Remote Bootstrap should update their scripts to reflect the new names.
- The SparkSQL datasource for Kudu no longer supports mode `Overwrite`. Users should use the new `KuduContext.upsertRows` method instead. Additionally, inserts using the datasource are now upserts by default. The older behavior can be restored by setting the `operation` parameter to `insert`.

Kudu 0.9.1 Release Notes

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

To upgrade Kudu to 0.9.1, see [Upgrade Parcels](#) or [Upgrade Packages](#).

Issues Fixed in Kudu 0.9.1

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

Kudu 0.9.0 Release Notes

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

To upgrade Kudu to 0.9.0, see [Upgrade Parcels](#) or [Upgrade Packages](#).

New Features in Kudu 0.9.0

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

Other Improvements and Changes in Kudu 0.9.0

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

Java

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

C++

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

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

- The default for `flush_threshold_mb` has been increased from 64 MB to 1000 MB.
- The default for `cfile_do_on_finish` has been changed from `close` to `flush`. [Experiments using YCSB](#) indicate that these values provide better throughput for write-heavy applications on typical server hardware.
- [KUDU-1415](#): Added statistics in the Java client, such as the number of bytes written and the number of operations applied.
- [KUDU-1451](#): Tablet servers take less time to restart when the tablet server must clean up many previously deleted tablets. Tablets are now cleaned up after they are deleted.

Issues Fixed in Kudu 0.9.0

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

Incompatible Changes in Kudu 0.9.0

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

Limitations of Kudu 0.9.0

Kudu 0.9.0 has the same limitations as Kudu 0.8, listed in [Limitations of Kudu 0.7.0](#) on page 35.

Upgrade Notes for Kudu 0.9.0

Before upgrading to Kudu 0.9.0, see [Incompatible Changes in Kudu 0.9.0](#) on page 32.

Kudu 0.8.0 Release Notes

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

To upgrade Kudu to 0.8.0, see [Upgrade Parcels](#) or [Upgrade Packages](#)

New Features in Kudu 0.8.0

- [KUDU-431](#): A simple Flume sink has been implemented.

Other Improvements in Kudu 0.8.0

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

Issues Fixed in Kudu 0.8.0

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

Incompatible Changes in Kudu 0.8.0

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

Limitations of Kudu 0.8.0

Kudu 0.8.0 has the same limitations as Kudu 0.7.0, listed in [Limitations of Kudu 0.7.0](#) on page 35.

Upgrade Notes for Kudu 0.8.0

Before upgrading to Kudu 0.8.0, see [Incompatible Changes in Kudu 0.8.0](#) on page 33.

Kudu 0.7.1 Release Notes

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

To upgrade Kudu to 0.7.1, see [Upgrade Parcels](#) or [Upgrade Packages](#).

Issues Fixed in Kudu 0.7.1

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

- [KUDU-1325](#) fixes a tablet server crash that could occur during table deletion. In some cases, while a table was being deleted, other replicas would attempt to re-replicate tablets to servers that had already processed the deletion. This could trigger a race condition that caused a crash.
- [KUDU-1341](#) fixes a potential data corruption and crash that could happen shortly after tablet server restarts in workloads that repeatedly delete and re-insert rows with the same primary key. In most cases, this corruption affected only a single replica and could be repaired by re-replicating from another.

- [KUDU-1343](#) fixes a bug in the Java client that occurs when a scanner has to scan multiple batches from one tablet and then start scanning from another. In particular, this affected any scans using the Java client that read large numbers of rows from multi-tablet tables.
- [KUDU-1345](#) fixes a bug where the hybrid clock could jump backwards, resulting in a crash followed by an inability to restart the affected tablet server.
- [KUDU-1360](#) fixes a bug in the `kudu-spark` module that prevented reading rows with `NULL` values.

Limitations of Kudu 0.7.1

Kudu 0.7.1 has the same limitations as Kudu 0.7.0, listed in [Limitations of Kudu 0.7.0](#) on page 35.

Upgrade Notes For Kudu 0.7.1

Kudu 0.7.1 has the same upgrade notes as Kudu 0.7.0, listed in [Upgrade Notes For Kudu 0.7.0](#) on page 36.

Kudu 0.7.0 Release Notes

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

To upgrade Kudu to 0.7, see [Upgrade Parcels](#) or [Upgrade Packages](#).

New Features in Kudu 0.7.0

Initial work for Spark integration

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

Other Improvements in Kudu 0.7.0

- Support for RHEL 7, CentOS 7, and SLES 12 has been added.
- The Python client is no longer considered experimental.
- The file block manager performance is improved, but it is still not recommended for real-world use.
- The master now attempts to spread tablets more evenly across the cluster during table creation. This has no impact on existing tables, but improves the speed at which under-replicated tablets are re-replicated after a tablet server failure.
- All licensing documents have been modified to adhere to ASF guidelines.
- The C++ client library is now explicitly built against the [old GCC 5 ABI](#). If you use `gcc5` to build a Kudu application, your application must use the old ABI as well. This is typically achieved by defining the `_GLIBCXX_USE_CXX11_ABI` macro at compile time when building your application. For more information, see [GCC 5 and the C++ 11 ABI](#).

Issues Fixed in Kudu 0.7.0

For a list of issues fixed in Kudu 0.7, see this [JIRA query](#).

Incompatible Changes in Kudu 0.7.0

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



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

- The default replication factor has been changed from 1 to 3. Existing tables continue to use the replication factor they were created with. Applications that create tables may not work properly if they assume a replication factor

of 1 and fewer than 3 replicas are available. To use the previous default replication factor, start the master with the configuration flag `--default_num_replicas=1`.

- The Python client has been rewritten, with a focus on improving code quality and testing. The read path (scanners) has been improved by adding many of the features already supported by the C++ and Java clients. The Python client is no longer considered experimental.

Limitations of Kudu 0.7.0

Operating System Limitations

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

Storage Limitations

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

Schema Limitations

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

Ingest Limitations

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

Cloudera Manager Limitations

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

Replication and Backup Limitations

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

Impala Limitations

- To use Kudu with Impala, you must install a special release of Impala. Obtaining and installing a compatible Impala release is detailed in [Using Apache Impala with Kudu](#) on page 73.
- To use Impala_Kudu alongside an existing Impala instance, you must install using parcels.

Apache Kudu Release Notes

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

Security Limitations

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

Client and API Limitations

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

Application Integration Limitations

- The Spark DataFrame implementation is not yet complete.

Other Known Issues

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

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

Upgrade Notes For Kudu 0.7.0

- Kudu 0.7.0 maintains wire compatibility with Kudu 0.6.0. A Kudu 0.7.0 client can communicate with a Kudu 0.6.0 cluster, and vice versa. For that reason, you do not need to upgrade client JARs at the same time the cluster is upgraded.
- The same wire compatibility guarantees apply to the `Impala_Kudu` fork that was released with Kudu 0.5.0.
- Review [Incompatible Changes in Kudu 0.7.0](#) on page 34 before upgrading to Kudu 0.7.

See [Upgrading Kudu](#) for instructions.

Kudu 0.6 Release Notes

To upgrade Kudu to 0.6, see [Upgrade Parcels](#) or [Upgrade Packages](#).

New Features in Kudu 0.6

Row Error Reporting

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

New Server-Side Metrics

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

Issues Fixed in Kudu 0.6

For a list of issues addressed in Kudu 0.6, see this [JIRA query](#).

Limitations of Kudu 0.6

Operating System Limitations

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

Storage Limitations

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

Schema Limitations

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

Ingest Limitations

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

Cloudera Manager Limitations

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

Replication and Backup Limitations

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

Impala Limitations

- To use Kudu with Impala, you must install a special release of Impala. Obtaining and installing a compatible Impala release is detailed in [Using Apache Impala with Kudu](#) on page 73.
- To use Impala_Kudu alongside an existing Impala instance, you must install using parcels.
- Updates, inserts, and deletes using Impala are non-transactional. If a query fails part of the way through, its partial effects will not be rolled back.
- All queries will be distributed across all Impala nodes which host a replica of the target table(s), even if a predicate on a primary key could correctly restrict the query to a single tablet. This limits the maximum concurrency of short queries made using Impala.
- No timestamp and decimal type support.
- The maximum parallelism of a single query is limited to the number of tablets in a table. For good analytic performance, aim for 10 or more tablets per host or large tables.
- Impala is only able to push down predicates involving `=`, `<=`, `>=`, or `BETWEEN` comparisons between a column and a literal value. Impala pushes down predicates `<` and `>` for integer columns only. For example, for a table with an integer key `ts`, and a string key `name`, the predicate `WHERE ts >= 12345` will convert into an efficient range scan, whereas `WHERE name > smith` will currently fetch all data from the table and evaluate the predicate within Impala.

Security Limitations

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

Client and API Limitations

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

Application Integration Limitations

- The Spark DataFrame implementation is not yet complete.

Other Known Issues

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

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

Upgrade Notes For Kudu 0.6

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

See [Upgrading Kudu](#) for instructions.

Kudu 0.5 Release Notes

Limitations of Kudu 0.5

Operating System Limitations

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

Storage Limitations

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

Schema Limitations

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

Ingest Limitations

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

Cloudera Manager Limitations

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

Replication and Backup Limitations

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

Impala Limitations

- To use Kudu with Impala, you must install a special release of Impala. Obtaining and installing a compatible Impala release is detailed in [Using Apache Impala with Kudu](#) on page 73.
- To use Impala_Kudu alongside an existing Impala instance, you must install using parcels.
- Updates, inserts, and deletes using Impala are non-transactional. If a query fails part of the way through, its partial effects will not be rolled back.

Apache Kudu Release Notes

- All queries will be distributed across all Impala nodes which host a replica of the target table(s), even if a predicate on a primary key could correctly restrict the query to a single tablet. This limits the maximum concurrency of short queries made using Impala.
- No timestamp and decimal type support.
- The maximum parallelism of a single query is limited to the number of tablets in a table. For good analytic performance, aim for 10 or more tablets per host or large tables.
- Impala is only able to push down predicates involving `=`, `<=`, `>=`, or `BETWEEN` comparisons between a column and a literal value. Impala pushes down predicates `<` and `>` for integer columns only. For example, for a table with an integer key `ts`, and a string key `name`, the predicate `WHERE ts >= 12345` will convert into an efficient range scan, whereas `WHERE name > smith` will currently fetch all data from the table and evaluate the predicate within Impala.

Security Limitations

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

Client and API Limitations

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

Application Integration Limitations

- The Spark DataFrame implementation is not yet complete.

Other Known Issues

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

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

Next Steps

[Kudu Quickstart](#)

[Installing Kudu](#)

[Configuring Kudu](#)

Apache Kudu Schema Design and Usage Limitations

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

Schema Design Limitations

Primary Key

- The 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.

Rows

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

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 the section on [Impala Integration Limitations](#) on page 43 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. See [Apache Kudu Schema Design](#) on page 89 for more information.
- Data in existing tables cannot currently be automatically repartitioned. As a workaround, create a new table with the new partitioning and insert the contents of the old table.
- 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 4TB.
- Recommended maximum number of tablets per tablet server is 1000, post-replication.
- Maximum number of tablets per table for each tablet server is 60, post-replication, at table-creation time.

Server Management Limitations

- Production deployments should configure a least 4GB of memory for tablet servers, and ideally more than 10GB.
- 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.

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

- 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

- Disk encryption is not built in. Kudu has been reported to run correctly on systems using local block device encryption (e.g. `dmccrypt`).
- 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.

- Kudu client instances retrieve authentication tokens upon first contact with the cluster. However, these tokens expire after one week and Kudu clients do not automatically request fresh tokens after initial token expiration. Therefore, Kudu clients that are active for more than a week are not supported.

Use of a single Kudu client instance for more than one week is only supported by the C++ client, not by the Java client.

Note that applications such as Apache Impala construct new clients for each query. Therefore, this limitation only affects the runtime of a single query.

Known Issues

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

Timeout Possible with Log Force Synchronization Option

If the Kudu master is configured with the `-log_force_fsync_all` option, tablet servers and clients will experience frequent timeouts, and the cluster may become unusable.

Longer Startup Times with a Large Number of Tablets

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

Kerberos Authentication Issues

- Kerberos authentication does not function correctly on hosts which contain capital letters in their hostname.
- Kerberos authentication does not function correctly if `rdns = false` is configured in `krb5.conf`.

Confusing Descriptions for Kudu TLS/SSL Settings in Cloudera Manager

Descriptions in the Cloudera Manager Admin Console for TLS/SSL settings are confusing, and will be replaced in a future release. The correct information about the settings is in the Usage Notes column:

Field	Usage Notes
Kerberos Principal	Set to the default principal, <code>kudu</code> .
Enable Secure Authentication And Encryption	Select this checkbox to enable authentication <i>and</i> RPC encryption between all Kudu clients and servers, as well as between individual servers. Only enable this property after you have configured Kerberos.
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.
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.

Field	Usage Notes
Master TLS/SSL Server CA Certificate (PEM Format)	Disregard this field.
Tablet Server TLS/SSL Server CA Certificate (PEM Format)	Disregard this field.
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.

Installing and Upgrading Apache Kudu



Important: This is the documentation for Apache Kudu 1.4.0 / CDH 5.12.x. Documentation for Apache Kudu 1.5.0 / CDH 5.13.x (and higher) is available as part of the [Cloudera Enterprise documentation](#).

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

Tip: To start using Kudu in minutes, without the need to install anything, see the [Kudu Quickstart](#) documentation.

Kudu Installation Requirements

- **Hardware**
 - Kudu currently requires a CPU that supports the SSSE3 and SSE4.2 instruction sets.
 - One or more hosts to run Kudu masters. You should have either one master (provides no fault tolerance), three masters (can tolerate one failure), or five masters (can tolerate two failures).
 - One or more hosts to run Kudu tablet servers. With replication, a minimum of three tablet servers is necessary.
- **Operating systems**
 - **Linux**
 - RHEL/CentOS 6.4, 6.5, 6.6, 6.7, 6.8, 7.1, 7.2, 7.3
 - Oracle Linux (OL) 6.4, 6.5, 6.6, 6.7, 6.8, 6.9, 7.1, 7.2, 7.3
 - Ubuntu 14.04 (Trusty), 16.04 (Xenial)
 - Debian 8.2, 8.4 (Jessie)
 - SLES 12 Service Pack 1
 - A kernel version and filesystem that support hole punching. Hole punching is the use of the `fallocate(2)` system call with the `FALLOC_FL_PUNCH_HOLE` option set. See [Error during hole punch test](#) on page 102. If you cannot meet this requirement, use [this workaround](#).
 - NTP
 - xfs or ext4 formatted drives.
 - **MacOS**
 - OS X 10.10 Yosemite, OS X 10.11 El Capitan, and macOS Sierra.
 - Pre-built macOS packages are not provided.
 - **Windows**
 - Microsoft Windows is not supported.
- **Management** - To manage Kudu with Cloudera Manager, Cloudera Manager 5.10.0 or later and CDH 5.10.0 or later are required.



Note: Kudu is not supported in Cloudera Manager's [single-user mode](#).

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

Install Kudu Using Cloudera Manager

You can install Kudu on a Cloudera Manager deployment using either parcels or packages.

Install Kudu Using Parcels

Starting with Kudu 1.3.0 / CDH 5.11.0, the Kudu CSD is included in Cloudera Manager. Manual installation of the Kudu CSD is not required. Use the following steps to install Kudu using parcels.

1. In Cloudera Manager, go to **Hosts > Parcels**. Find `KUDU` in the list, and click **Download**.
2. When the download is complete, select your cluster from the **Locations** selector, and click **Distribute**. If you only have one cluster, it is selected automatically.
3. When distribution is complete, click **Activate** to activate the parcel. Restart the cluster when prompted. This might take several minutes.
4. Install the Kudu service on your cluster. Go to the cluster where you want to install Kudu. Click **Actions > Add a Service**. Select **Kudu** from the list, and click **Continue**.
5. Select a host for the master role and one or more hosts for the tablet server roles. A host can act as both a master and a tablet server, but this might cause performance problems on a large cluster. The Kudu master process is not resource-intensive and can be collocated with other similar processes such as the HDFS NameNode or YARN ResourceManager. After selecting hosts, click **Continue**.
6. Configure the storage locations for Kudu data and write-ahead log (WAL) files on masters and tablet servers. Cloudera Manager will create the directories.
 - You can use the same directory to store data and WALs.
 - You cannot store WALs in a subdirectory of the data directory.
 - If any host is both a master and tablet server, configure different directories for master and tablet server. For instance, `/data/kudu/master` and `/data/kudu/tserver`.
 - If you have chosen a filesystem that does not support hole punching, the Kudu service will fail to start. In this case *only*, exit the wizard by clicking the Cloudera logo at the top left, and enable the file block manager. This is not appropriate for production. See [Enabling the File Block Manager](#) on page 47.
7. If your filesystem supports hole punching, do not exit the wizard. Click **Continue**. Kudu masters and tablet servers are started. Otherwise, go to the **Kudu** service, and click **Actions > Start**.
8. [Verify the Installation](#) on page 51.
9. To manage roles, go to the **Kudu** service and use the **Actions** menu to stop, start, restart, or otherwise manage the service.

Enabling the File Block Manager

If your filesystem supports hole punching, do not use the file blocker manager. The file blocker manager does not perform well at scale and must only be used for small-scale development and testing.

If your filesystem does not support hole punching, but you want to experiment with Kudu, you must enable the file block manager. If you do not enable the file block manager, Kudu will not start.

1. If you are still in the Cloudera configuration wizard, exit the configuration wizard by clicking the Cloudera logo at the top of the Cloudera Manager interface.
2. Go to the **Kudu** service.
3. Go to **Configuration** and search for the **Kudu Service Advanced Configuration Snippet (Safety Valve) for gflagfile** configuration option.
4. Add the following line to it, and save your changes:

```
--block_manager=file
```

Install Kudu Using Packages

Table 1: Kudu Repository and Package Links

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

1. Cloudera recommends installing the Kudu repositories for your operating system. Use the links in [Table 1: Kudu Repository and Package Links](#) on page 48 to download the appropriate repository installer. Save the repository installer to `/etc/yum.repos.d/` for RHEL, `/etc/apt/sources.list.d/` for Ubuntu/Debian, or `/etc/zypp/repos.d` for SLES.
2. Add the Cloudera Public GPG repository key for each operating system in the cluster. This key enables you to verify that you are downloading genuine packages.

Operating System	Command
RHEL/CentOS 6	<pre>sudo rpm --import https://archive.cloudera.com/kudu/redhat/6/x86_64/kudu/RPM-GPG-KEY-cloudera</pre>
RHEL/CentOS 7	<pre>sudo rpm --import https://archive.cloudera.com/kudu/redhat/7/x86_64/kudu/RPM-GPG-KEY-cloudera</pre>
SLES	<pre>sudo rpm --import https://archive.cloudera.com/kudu/sles/12/x86_64/kudu/RPM-GPG-KEY-cloudera</pre>
Debian	<p>Jessie</p> <pre>wget https://archive.cloudera.com/kudu/debian/jessie/amd64/kudu/archive.key -O archive.key sudo apt-key add archive.key</pre>
Ubuntu	<p>Xenial</p> <pre>wget https://archive.cloudera.com/kudu/ubuntu/xenial/amd64/kudu/archive.key -O archive.key sudo apt-key add archive.key</pre> <p>Trusty</p> <pre>wget https://archive.cloudera.com/kudu/ubuntu/trusty/amd64/kudu/archive.key -O archive.key sudo apt-key add archive.key</pre>

3. Install the Kudu packages.
 - If you use Cloudera Manager, you only need to install the `kudu` package:

Operating System	Install Commands
RHEL/CentOS	<pre>sudo yum install kudu</pre>
Ubuntu/Debian	<pre>sudo apt-get install kudu</pre>
SLES	<pre>sudo zypper install kudu</pre>

- If you need the C++ client development libraries or the Kudu SDK, install `kudu-client` and `kudu-client-devel` packages for RHEL, or `libkuduclient0` and `libkuduclient-dev` packages for Ubuntu.
 - **Do not** install the `kudu-master` and `kudu-tserver` packages. They provide operating system startup scripts for using Kudu without Cloudera Manager.
4. Install the Kudu service on your cluster. Go to the cluster where you want to install Kudu. Click **Actions > Add a Service**. Select **Kudu** from the list, and click **Continue**.
 5. Select a host for the master role and one or more hosts for the tablet server roles. A host can act as both a master and a tablet server, but this might cause performance problems on a large cluster. The Kudu master process is not resource-intensive and can be collocated with other similar processes such as the HDFS NameNode or YARN ResourceManager. After selecting hosts, click **Continue**.
 6. Configure the storage locations for Kudu data and write-ahead log (WAL) files on masters and tablet servers. Cloudera Manager will create the directories.
 - You can use the same directory to store data and WALs.
 - You cannot store WALs in a subdirectory of the data directory.
 - If any host is both a master and tablet server, configure different directories for master and tablet server. For instance, `/data/kudu/master` and `/data/kudu/tserver`.
 - If you have chosen a filesystem that does not support hole punching, the Kudu service will fail to start. In this case *only*, exit the wizard by clicking the Cloudera logo at the top left, and enable the file block manager. This is not appropriate for production. See [Enabling the File Block Manager](#) on page 47.
 7. If your filesystem supports hole punching, do not exit the wizard. Click **Continue**. Kudu masters and tablet servers are started. Otherwise, go to the **Kudu** service, and click **Actions > Start**.
 8. [Verify the Installation](#) on page 51.
 9. To manage roles, go to the **Kudu** service and use the **Actions** menu to stop, start, restart, or otherwise manage the service.

Install Kudu Using the Command Line



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

Follow these steps on each node in your Kudu cluster.

1. Cloudera recommends installing the Kudu repositories for your operating system. Use the links in the following table to download the appropriate repository installer. Save the repository installer to `/etc/yum/repos.d/` for RHEL, `/etc/apt/sources.list.d/` for Ubuntu/Debian, or `/etc/zypp/repos.d` for SLES.

Table 2: Kudu Repository and Package Links

Operating System	Repository Package	Individual Packages
RHEL	RHEL 6 or RHEL 7	RHEL 6

Operating System	Repository Package	Individual Packages
Ubuntu	Trusty , Xenial	Trusty , Xenial
SLES	SLES 12	SLES 12
Debian	Jessie	Jessie

2. Add the Cloudera Public GPG repository key for each operating system in the cluster. This key enables you to verify that you are downloading genuine packages.

Operating System	Command
RHEL/CentOS 6	<pre>sudo rpm --import https://archive.cloudera.com/kudu/redhat/6/x86_64/kudu/RPM-GPG-KEY-cloudera</pre>
RHEL/CentOS 7	<pre>sudo rpm --import https://archive.cloudera.com/kudu/redhat/7/x86_64/kudu/RPM-GPG-KEY-cloudera</pre>
SLES	<pre>sudo rpm --import https://archive.cloudera.com/kudu/sles/12/x86_64/kudu/RPM-GPG-KEY-cloudera</pre>
Debian	<p>Jessie</p> <pre>wget https://archive.cloudera.com/kudu/debian/jessie/amd64/kudu/archive.key -O archive.key sudo apt-key add archive.key</pre>
Ubuntu	<p>Xenial</p> <pre>wget https://archive.cloudera.com/kudu/ubuntu/xenial/amd64/kudu/archive.key -O archive.key sudo apt-key add archive.key</pre> <p>Trusty</p> <pre>wget https://archive.cloudera.com/kudu/ubuntu/trusty/amd64/kudu/archive.key -O archive.key sudo apt-key add archive.key</pre>

3. Install the `kudu` package, using the appropriate commands for your operating system. Also install the `kudu-master` and `kudu-tserver` packages. They provide operating system start-up scripts for the Kudu master and tablet servers.

Operating System	Install Commands
RHEL/CentOS	<pre> sudo yum install kudu # Base Kudu files sudo yum install kudu-master # Kudu master init.d service script and default configuration sudo yum install kudu-tserver # Kudu tablet server init.d service script and default configuration sudo yum install kudu-client0 # Kudu C++ client shared library sudo yum install kudu-client-devel # Kudu C++ client SDK </pre>
Ubuntu/Debian	<pre> sudo apt-get install kudu # Base Kudu files sudo apt-get install kudu-master # Service scripts for managing kudu-master sudo apt-get install kudu-tserver # Service scripts for managing kudu-tserver sudo apt-get install libkuduclient0 # Kudu C++ client shared library sudo apt-get install libkuduclient-dev # Kudu C++ client SDK </pre>
SLES	<pre> sudo zypper install kudu # Base Kudu files sudo zypper install kudu-master # Kudu master init.d service script and default configuration sudo zypper install kudu-tserver # Kudu tablet server init.d service script and default configuration sudo zypper install kudu-client0 # Kudu C++ client shared library sudo zypper install kudu-client-devel # Kudu C++ client SDK </pre>

- The packages create a `kudu-conf` entry in the operating system's alternatives database, and they ship the built-in `conf.dist` alternative. To adjust your configuration, you can either edit the files in `/etc/kudu/conf/` directly, or create a new alternative using the operating system utilities. If you create a new alternative, make sure the alternative is the directory pointed to by the `/etc/kudu/conf/` symbolic link, and create custom configuration files there. Some parts of the configuration are configured in `/etc/default/kudu-master` and `/etc/default/kudu-tserver` files as well. You must include or duplicate these configuration options if you create custom configuration files.

Review the configuration, including the default WAL and data directory locations, and adjust them according to your requirements.

- 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 / Debian
sudo update-rc.d kudu-tserver defaults # Ubuntu / Debian

```

For instructions on how to perform common administrative tasks in Kudu, see [Apache Kudu Administration](#) on page 57.

- [Verify the Installation](#) on page 51.

Verify the Installation

- Verify that the Kudu master and tablet servers are running using one of the following methods:

Installing and Upgrading Apache Kudu

- Examine the output of the `ps` command on servers to verify that the `kudu-master` and `kudu-tserver` processes are running.
 - Access the master or tablet server web UI by going to `http://<_host_name_>:8051/` for masters, or `http://<_host_name_>:8050/` for tablet servers.
2. If Kudu isn't running, look at the log files in `/var/log/kudu`, and if there's a file ending with `.FATAL`, that means Kudu did not start.
 - If the error is related to a failed hole punch test or the file block manager, it might be a problem [with your operating system](#).
 - If the error is related to clock synchronization, it is most likely a [problem with the Network Time Protocol](#).

Upgrade Kudu using Cloudera Manager



Warning:

- You must upgrade Kudu *before* you upgrade CDH. Failure to do so will result in Impala queries returning the following error:

```
Query Status: Unable to open scanner: Remote error: unsupported feature flags
```

- 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.

To use Cloudera Manager to upgrade Kudu using parcels or packages, use the following instructions. If you do not use Cloudera Manager, see [Upgrade Kudu Using the Command Line](#) on page 53.

Before upgrading Kudu, read the [Release Notes](#) relevant to the version you are upgrading to.

Upgrade Kudu Using Parcels

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

Upgrade Kudu Using Packages

1. If you use a repository, re-download the repository list file to ensure that you have the latest information. See [Table 1: Kudu Repository and Package Links](#) on page 48.
2. Stop the Kudu service in Cloudera Manager. Go to the Kudu service and select **Actions > Stop**.
3. Depending on your operating system, issue the following set of commands on each Kudu host:

Operating System	Upgrade Commands
RHEL/CentOS	<pre>sudo yum -y clean all sudo yum -y upgrade kudu</pre>
Ubuntu/Debian	<pre>sudo apt-get update sudo apt-get install kudu</pre>
SLES	<pre>sudo zypper clean --all sudo zypper update kudu</pre>

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

Upgrade Kudu Using the Command Line



Warning:

- You must upgrade Kudu *before* you upgrade CDH. Failure to do so will result in Impala queries returning the following error:

```
Query Status: Unable to open scanner: Remote error: unsupported feature flags
```

- 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.

If you use Cloudera Manager, do not use the following command-line instructions. See [Upgrade Kudu using Cloudera Manager](#) on page 52.

Before upgrading Kudu, read the [Release Notes](#) relevant to the version you are upgrading to. Note that rolling upgrades are not supported. Shut down all Kudu services before you begin upgrading the software.

1. If you use a repository, re-download the repository list file to ensure that you have the latest information. See [Table 1: Kudu Repository and Package Links](#) on page 48.
2. Stop the Kudu master and tablet servers using the following commands:

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

3. Depending on your operating system, issue the following set of commands on each Kudu host:

Operating System	Upgrade Commands
RHEL/CentOS	<pre>sudo yum -y clean all sudo yum -y upgrade kudu</pre>
Ubuntu/Debian	<pre>sudo apt-get update sudo apt-get install kudu</pre>
SLES	<pre>sudo zypper clean --all sudo zypper update kudu</pre>

4. Start the Kudu master and tablet servers using the following commands:

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

Next Steps

Read about [Using Apache Impala with Kudu](#) on page 73.

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

For a reading list and other helpful links, refer to [More Resources for Apache Kudu](#) on page 107.

Apache Kudu Configuration



Important: This is the documentation for Apache Kudu 1.4.0 / CDH 5.12.x. Documentation for Apache Kudu 1.5.0 / CDH 5.13.x (and higher) is available as part of the [Cloudera Enterprise documentation](#).

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 3: Supported Configuration Flags for Kudu Masters

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

Flag	Valid Options	Default	Description
			--fs_data_dirs, but not a sub-directory of a data directory.
--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 4: Supported Configuration Flags for Kudu Tablet Servers

Flag	Valid Options	Default	Description
--fs_data_dirs	string		Comma-separated list of directories where the Tablet Server will place its data blocks.
--fs_wal_dir	string		The directory where the Tablet Server will place its write-ahead logs. Can be the same as <i>one of</i> the directories listed in <code>--fs_data_dirs</code> , but not a sub-directory of a data directory.
--log_dir	string	/tmp	The directory to store Tablet Server log files
--tserver_master_addrs	string	127.0.0.1:7051	Comma separated addresses of the masters 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



Important: This is the documentation for Apache Kudu 1.4.0 / CDH 5.12.x. Documentation for Apache Kudu 1.5.0 / CDH 5.13.x (and higher) is available as part of the [Cloudera Enterprise documentation](#).

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.

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.



Important:

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

Prepare for the migration

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

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

master_data_dir

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

For example:

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

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



Important:

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

4. Perform the following preparatory steps for each new master:

- Choose an unused machine in the cluster. The master generates very little load so it can be collocated with other data services or load-generating processes, though not with another Kudu master from the same configuration.
- Ensure Kudu is installed on the machine, either using system packages (in which case the `kudu` and `kudu-master` packages should be installed), or some other means.
- Choose and record the directory where the master's data will live.
- Choose and record the port the master should use for RPCs.
- **(Optional)** Configure a DNS alias for the master (e.g. `master-2`, `master-3`, etc).

Perform the migration

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

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

master_data_dir

The new master's previously recorded data directory.

For example:

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

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

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

master_data_dir

The existing master's previously recorded data directory

tablet_id

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

all_masters

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

uuid

The master's previously recorded UUID.

hostname

The master's previously recorded hostname or alias.

port

The master's previously recorded RPC port number.

For example:

```
$ kudu local_replica cmeta rewrite_raft_config --fs_wal_dir=/var/lib/kudu/master
00000000000000000000000000000000 4aab798a69e94fab8d77069edff28ce0:master-1:7051
f5624e05f40649b79a757629a69d061e:master-2:7051
988d8ac6530f426cbe180be5ba52033d:master-3:7051
```

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

hostname

The master's previously recorded hostname or alias.

port

The master's previously recorded RPC port number.

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

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

master_data_dir

The new master's previously recorded data directory.

tablet_id

Must be set to the string, `00000000000000000000000000000000`.

existing_master

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

hostname

The existing master's previously recorded hostname or alias.


port

The existing master's previously recorded RPC port number.

Example

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

8. Start all the new masters.



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

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

hostname

The master's previously recorded hostname or alias

port

The master's previously recorded RPC port number

- 10 Start all the tablet servers.

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

- Using a browser, visit each master's web UI and navigate to the `/masters` page. All the masters should now be listed there with one master in the `LEADER` role and the others in the `FOLLOWER` role. The contents of `/masters` on each master should be the same.
- Run a Kudu system check (`ksck`) on the cluster using the `kudu` command line tool. For more details, see [Monitoring Cluster Health with ksck](#) on page 65.

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

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

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



Important:

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

Prepare for the recovery

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

- Identify and record the master’s UUID. It can be fetched using the following command:

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

master_data_dir

live master’s previously recorded data directory

Example

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

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

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

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

master_data_dir

The reference master’s previously recorded data directory.

tablet_id

Must be set to the string, 00000000000000000000000000000000.

Example

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

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

Perform the recovery

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

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

master_data_dir

The replacement master’s previously recorded data directory.

uuid

The dead master’s previously recorded UUID.

For example:

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


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

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

master_data_dir

The replacement master's previously recorded data directory.

tablet_id

Must be set to the string, 00000000000000000000000000000000.

reference_master

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

hostname

The reference master's previously recorded hostname or alias.

port

The reference master's previously recorded RPC port number.

For example:

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

3. If you are using Cloudera Manager, add the replacement Kudu master role now, but do not start it.

- Override the empty value of the `Master Address` parameter for the new role with the replacement master's alias.
- If you are using a non-default RPC port, add the port number (separated by a colon) as well.

4. Reconfigure the DNS alias for the dead master to point to the replacement master.

5. Start the replacement master.

6. Restart the existing live masters. This results in a brief availability outage, but it should last only as long as it takes for the masters to come back up.

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

- Using a browser, visit each master's web UI and navigate to the `/masters` page. All the masters should now be listed there with one master in the `LEADER` role and the others in the `FOLLOWER` role. The contents of `/masters` on each master should be the same.
- Run a Kudu system check (`ksck`) on the cluster using the `kudu` command line tool. For more details, see [Monitoring Cluster Health with ksck](#) on page 65.

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:

```
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 visit the [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)

WARNING: 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:

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

Recovering from Disk Failure

Kudu tablet servers are not resistant to disk failure. When a disk containing a data directory or the write-ahead log (WAL) dies, the entire tablet server must be rebuilt. Kudu will automatically re-replicate tablets on other servers after a tablet server fails, but manual intervention is needed in order to restore the failed tablet server to a running state.

The first step to restoring a tablet server after a disk failure is to replace the failed disk, or remove the failed disk from the data-directory and/or WAL configuration. Next, the existing data directories and WAL directory must be removed. For example, if the 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 existing data directories and WAL directory:

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

After the WAL and data directories are removed, the tablet server process can be started. When Kudu is installed using system packages, `service` is typically used as follows:

```
sudo service kudu-tserver start
```

Once the tablet server is running again, new tablet replicas will be created on it as necessary.

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 65.
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 65.
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.

Developing Applications With Apache Kudu



Important: This is the documentation for Apache Kudu 1.4.0 / CDH 5.12.x. Documentation for Apache Kudu 1.5.0 / CDH 5.13.x (and higher) is available as part of the [Cloudera Enterprise documentation](#).

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`.

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 protocbuf
```

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 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](#).

Kudu Python Client

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

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

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

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

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

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

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

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

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

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

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

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

# Flush write operations, if failures occur, capture print them.
try:
    session.flush()
except kudu.KuduBadStatus as e:
    print(session.get_pending_errors())

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

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

Example Apache Impala Commands With Kudu

See [Using Apache Impala with Kudu](#) on page 73 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")

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

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

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

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

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

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

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

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

Spark Integration Known Issues and Limitations

- Kudu tables with a name containing upper case or non-ASCII characters must be assigned an alternate name when registered as a temporary table.
- Kudu tables with a column name containing upper case or non-ASCII characters 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



Important: This is the documentation for Apache Kudu 1.4.0 / CDH 5.12.x. Documentation for Apache Kudu 1.5.0 / CDH 5.13.x (and higher) is available as part of the [Cloudera Enterprise documentation](#).

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



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

Internal and External Impala Tables

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

Internal

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

External

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

See the [Impala documentation](#) for more information about internal and external tables.

Using Impala To Query Kudu Tables

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

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

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

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

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

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

- Start Impala Shell using the `impala-shell` command. By default, `impala-shell` attempts to connect to the Impala daemon on `localhost` on port 21000. To connect to a different host, use the `-i <host:port>` option. To automatically connect to a specific Impala database, use the `-d <database>` option. For instance, if all your Kudu tables are in Impala in the database `impala_kudu`, use `-d impala_kudu` to use this database.
- To quit the Impala Shell, use the following command: `quit;`

Querying an Existing Kudu Table from Impala

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

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

Creating a New Kudu Table From Impala

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

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

When creating a new table in Kudu, you *must define a partition schema* to pre-split your table. The best partition schema to use depends upon the structure of your data and your data access patterns. The goal is to maximize parallelism and use all your tablet servers evenly. For more information on partition schemas, see [Partitioning Tables](#) on page 75.

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

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

CREATE TABLE AS SELECT

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

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

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

Partitioning Tables

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

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

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

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



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

Basic Partitioning

PARTITION BY RANGE

You can specify range partitions for one or more primary key columns. Range partitioning in Kudu allows splitting a table based 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 77 for an extended example.

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

```

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

```

Advanced Partitioning

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

`PARTITION BY HASH` and `RANGE`

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

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

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

```

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

```

Multiple `PARTITION BY HASH` Definitions

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

```

CREATE TABLE cust_behavior (
  id BIGINT,

```

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

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

Non-Covering Range Partitions

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

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



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

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

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

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

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

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

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

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

Partitioning Guidelines

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

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

Optimizing Performance for Evaluating SQL Predicates

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

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

Inserting a Row

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

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

The primary key must not be null.

Inserting In Bulk

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

Multiple Single `INSERT` statements

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

Single `INSERT` statement with multiple `VALUES` subclauses

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

```
set batch_size=10000;
```



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

Batch Insert

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

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

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

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

Ingest using the C++ or Java API

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

INSERT and Primary Key Uniqueness Violations

In many relational databases, if you try to insert a row that has already been inserted, the insertion will fail because the primary key will be duplicated (see [Failures During INSERT, UPDATE, UPSERT, and DELETE Operations](#) on page 82). 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 79.

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
```

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 79.

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 82.

```
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 [Apache Kudu Security](#) on page 84.

Sentry Authorization works as follows:

- Only the ALL privilege can be granted on Kudu tables. This means SELECT or INSERT grants are not supported.
- Column-level permissions are not supported.
- Only users with ALL privileges on SERVER may create external Kudu tables.

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](#).

Apache Kudu Security



Important: This is the documentation for Apache Kudu 1.4.0 / CDH 5.12.x. Documentation for Apache Kudu 1.5.0 / CDH 5.13.x (and higher) is available as part of the [Cloudera Enterprise documentation](#).

Kudu 1.3 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 43.

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.

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 > Security**.
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, <code>kudu</code> . Currently, Kudu does not support configuring a custom service principal for Kudu processes.
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 <code>impala</code> 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.
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.

Field	Usage Notes
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 86.

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



Important: This is the documentation for Apache Kudu 1.4.0 / CDH 5.12.x. Documentation for Apache Kudu 1.5.0 / CDH 5.13.x (and higher) is available as part of the [Cloudera Enterprise documentation](#).

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
<code>int8</code> , <code>int16</code> , <code>int32</code>	<code>plain</code> , <code>bitshuffle</code> , <code>run length</code>	<code>bitshuffle</code>
<code>int64</code> , <code>unixtime_micros</code>	<code>plain</code> , <code>bitshuffle</code> , <code>run length</code>	<code>bitshuffle</code>
<code>float</code> , <code>double</code>	<code>plain</code> , <code>bitshuffle</code>	<code>bitshuffle</code>
<code>bool</code>	<code>plain</code> , <code>run length</code>	<code>run length</code>
<code>string</code> , <code>binary</code>	<code>plain</code> , <code>prefix</code> , <code>dictionary</code>	<code>dictionary</code>

Column Compression

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

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

Primary Key Design

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

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

Primary Key Index

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

Partitioning

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

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



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

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

Range Partitioning

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

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

Adding and Removing Range Partitions

Kudu allows range partitions to be dynamically added and removed from a table at runtime, without affecting the availability of other partitions. Removing a partition will delete the tablets belonging to the partition, as well as the data contained in them. Subsequent inserts into the dropped partition will fail. New partitions can be added, but they

must not overlap with any existing range partitions. Kudu allows dropping and adding any number of range partitions in a single transactional alter table operation.

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

Hash Partitioning

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

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

Multilevel Partitioning

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

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

Partition Pruning

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

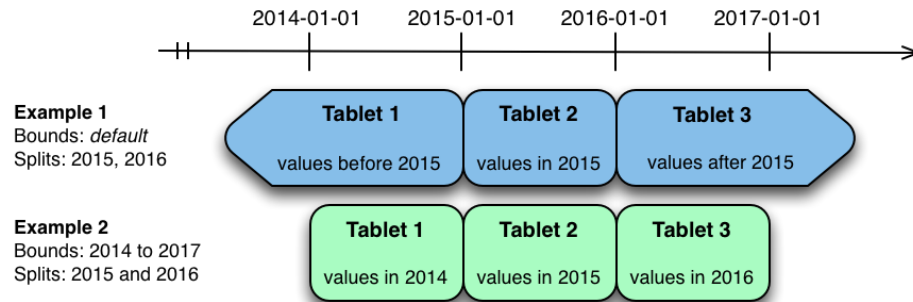
Partitioning Examples

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

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

Range Partitioning

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



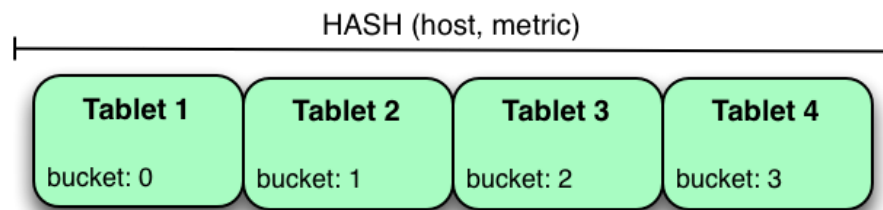
The image above shows the two ways the `metrics` table can be range partitioned on the `time` column. In the first example (in blue), the default range partition bounds are used, with splits at `2015-01-01` and `2016-01-01`. This results in three tablets: the first containing values before 2015, the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

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

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

Hash Partitioning

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



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

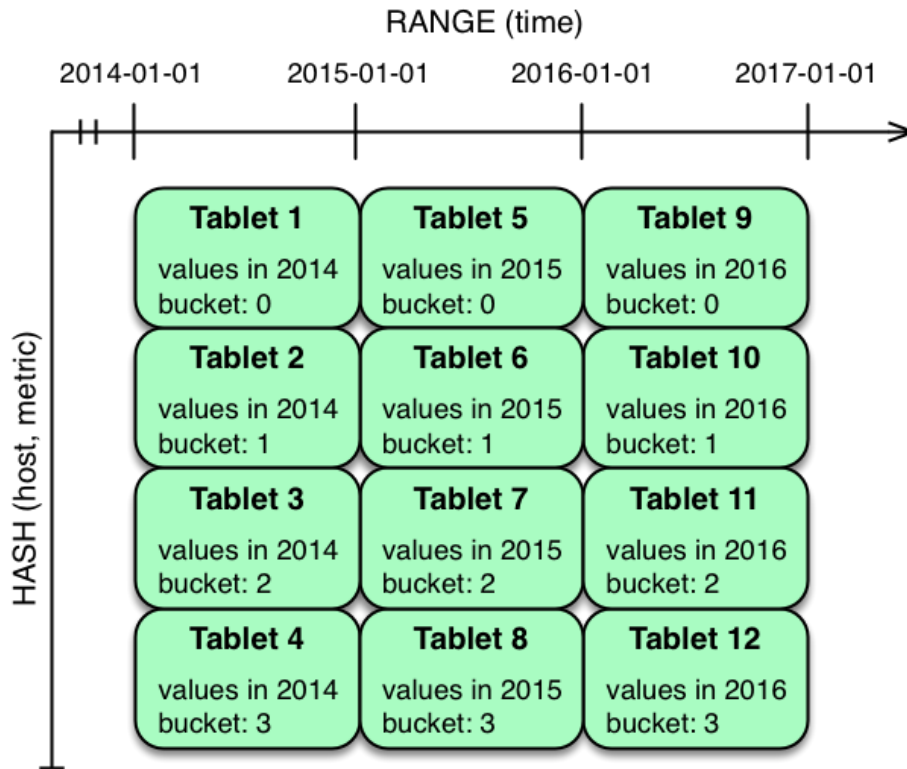
Hash and Range Partitioning

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

Table 5: 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

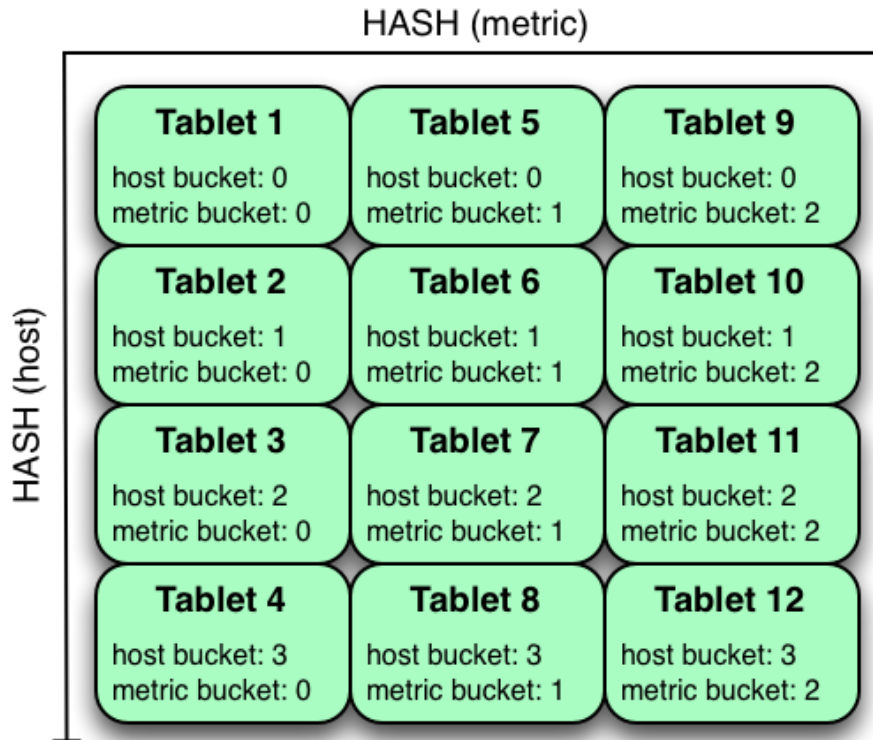
Hash partitioning is good at maximizing write throughput, while range partitioning avoids issues of unbounded tablet growth. Both strategies can take advantage of partition pruning to optimize scans in different scenarios. Using multilevel partitioning, it is possible to combine the two strategies in 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 Schema Design and Usage Limitations](#) on page 40.

Apache Kudu Transaction Semantics



Important: This is the documentation for Apache Kudu 1.4.0 / CDH 5.12.x. Documentation for Apache Kudu 1.5.0 / CDH 5.13.x (and higher) is available as part of the [Cloudera Enterprise documentation](#).

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

Kudu currently allows the following operations:

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

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

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

Single Tablet Write Operations

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

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

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

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

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

Writing to Multiple Tablets

Kudu does not support transactions that span multiple tablets. However, consistent snapshot reads are possible (with caveats, as explained below). Writes from a Kudu client are optionally buffered in memory until they are flushed and sent to the tablet server. When a client's session is flushed, the rows for each tablet are batched together, and sent to the tablet server which hosts the leader replica of the tablet. Since there are no inter-tablet transactions, each of

these batches represents a single, independent write operation with its own timestamp. However, the client API provides the option to impose some constraints on the assigned timestamps and on how writes to different tablets are observed by clients.

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

- **CLIENT_PROPAGATED Consistency**

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

Java Client

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

C++ Client

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

- **COMMIT_WAIT Consistency**

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

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

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

Java Client

Call `KuduSession#setExternalConsistencyMode(ExternalConsistencyMode.COMMIT_WAIT)`

C++ Client

Call `KuduSession::SetExternalConsistencyMode(COMMIT_WAIT)`



Warning:

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

Read Operations (Scans)

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

Java Client

Call `KuduScannerBuilder#setReadMode(...)`

C++ Client

Call `KuduScanner::SetReadMode()`

The following modes are available in both clients:

READ_LATEST

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

READ_AT_SNAPSHOT

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

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

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

Known Issues and Limitations

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

Reads (Scans)

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

Recommendation

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

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

Writes

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

Recommendation

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

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

Apache Kudu Background Maintenance Tasks



Important: This is the documentation for Apache Kudu 1.4.0 / CDH 5.12.x. Documentation for Apache Kudu 1.5.0 / CDH 5.13.x (and higher) is available as part of the [Cloudera Enterprise documentation](#).

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



Important: This is the documentation for Apache Kudu 1.4.0 / CDH 5.12.x. Documentation for Apache Kudu 1.5.0 / CDH 5.13.x (and higher) is available as part of the [Cloudera Enterprise documentation](#).

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

Issues Starting or Restarting the Master or Tablet Server

Error during hole punch test

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

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

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

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

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	<code>sudo apt-get install ntp</code>
RHEL/CentOS	<code>sudo yum install ntp</code>

- 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	<code>sudo service ntp restart</code>
RHEL/CentOS	<code>sudo /etc/init.d/ntpd restart</code>

- 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`.

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.

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:
[<ffffffffffa00b209d>] do_get_write_access+0x29d/0x520 [jbd2]
[<ffffffffffa00b2471>] jbd2_journal_get_write_access+0x31/0x50 [jbd2]
[<ffffffffffa00fe6d8>] __ext4_journal_get_write_access+0x38/0x80 [ext4]
[<ffffffffffa00d9b23>] ext4_reserve_inode_write+0x73/0xa0 [ext4]
[<ffffffffffa00d9b9c>] ext4_mark_inode_dirty+0x4c/0x1d0 [ext4]
[<ffffffffffa00d9e90>] ext4_dirty_inode+0x40/0x60 [ext4]
[<ffffffffff811ac48b>] __mark_inode_dirty+0x3b/0x160
[<ffffffffff8119c742>] file_update_time+0xf2/0x170
[<ffffffffff8111c1e0>] __generic_file_aio_write+0x230/0x490
[<ffffffffff8111c4c8>] generic_file_aio_write+0x88/0x100
[<ffffffffffa00d3fb1>] ext4_file_write+0x61/0x1e0 [ext4]
[<ffffffffffa00d3fb1>] ext4_file_write+0x61/0x1e0 [ext4]
[<ffffffffff81180f5b>] do_sync_readv_writev+0xfb/0x140
[<ffffffffff81181ee6>] 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.

More Resources for Apache Kudu



Important: This is the documentation for Apache Kudu 1.4.0 / CDH 5.12.x. Documentation for Apache Kudu 1.5.0 / CDH 5.13.x (and higher) is available as part of the [Cloudera Enterprise documentation](#).

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.