Cloudera Runtime 7.2.12

Apache Kudu Overview

Date published: 2020-02-28
Date modified: 2021-10-25

https://docs.cloudera.com/

Legal Notice

Contents

Kudu introduction 4

Kudu architecture in a CDP public cloud deployment 4

Kudu network architecture 5

Kudu-Impala integration 6

Example use cases 7

Kudu concepts 8

Apache Kudu usage limitations 10

- Schema design limitations 10
- Partitioning limitations 11
- Scaling recommendations and limitations 11
- Server management limitations 12
- Cluster management limitations 12
- Impala integration limitations 12
- Spark integration limitations 13
- Security limitations 13
- Other known issues 13

More resources for Apache Kudu 13

Kudu introduction

Apache Kudu is a columnar storage manager developed for the Hadoop platform. Kudu shares the common technical properties of Hadoop ecosystem applications: Kudu 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 consistenc
- 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 historical data

Kudu architecture in a CDP public cloud deployment

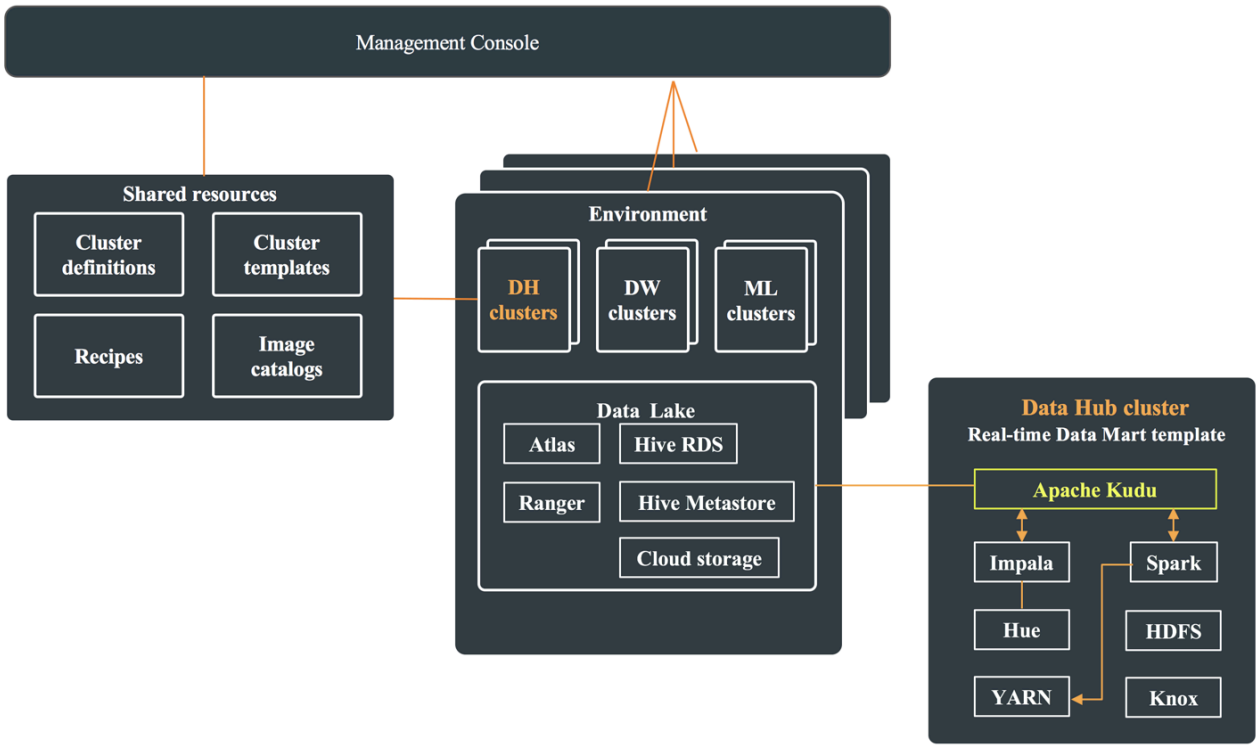
In a CDP public cloud deployment, Kudu is available as one of the many Cloudera Runtime services within the Real-time Data Mart template. To use Kudu, you can create a Data Hub cluster by selecting Real-time Data Mart template template in the Management Console.

Each Data Hub cluster that is deployed using the Real-time Data Mart template has an instance of Apache Kudu, Impala, Spark, and Knox. It also contains YARN—which is used to run Spark, Hue—which can be used to issue queries using Impala, and HDFS—because the cluster is managed using Cloudera Manager. These components use the shared resources present within the Data Lake.

Impala is primarily used to run analytical queries against the data stored in Kudu. It also manages hierarchical storage between Kudu and the object storage. You can use SQL statements to move older data in the range partitions from Kudu into the Parquet partitions on object storage, for example, on Amazon S3 as cold storage. You can then use an Impala UNION ALL or VIEW statement to query the Kudu and Parquet data together.

Besides ingesting data, Spark is used for backup and disaster recovery (BDR) operations. The Kudu BDR feature is built as a Spark application, and it can take either full or incremental table backups. It then restores a stack of backups into a table.

The following diagram shows Apache Kudu and its dependencies as a part of the Data Hub cluster, and the shared resources they use which are present in the Data Lake:



Kudu network architecture

The highly scalable feature of Kudu is a result of Kudu's distributed nature. This topic describes how the logical components within a Kudu cluster interact with each other to achieve high availability and fault tolerance.

Kudu runs on a cluster of machines. These machines collectively store and manage Kudu's data. The cluster architecture is designed for fault tolerance and high availability. You can add more nodes to the Kudu cluster as your storage and throughput needs increase.

Kudu's architecture is based on tablets. A tablet contains a subset of the data in a table. Instead of storing the data in a table in a single location, Kudu distributes the table's tablets to multiple machines in the cluster. Those machines store the tablet's data on their local disks and respond to client requests to read and write table data. A Kudu cluster consists of one or more master nodes and one or more tablet servers. Tablet servers do most of the work in the cluster and typically run on machines known as worker nodes. Tablet servers are responsible for storing and processing data.

Kudu masters don't serve or store table data. They are responsible for the cluster's metadata, such as the schema for every table and on which tablet servers each table's data is stored.

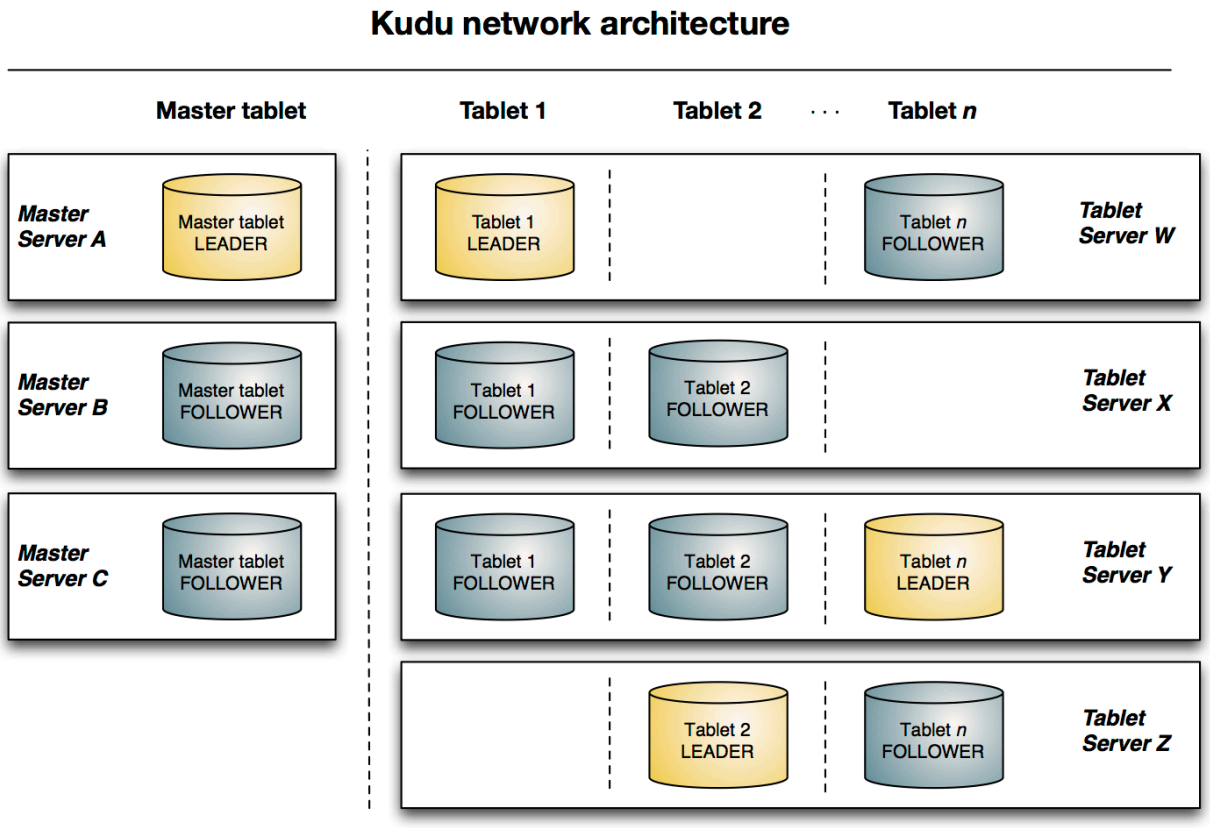
System administrators have the option of setting up a Kudu cluster to be highly available. Therefore, if a single node becomes unavailable, it doesn't cause the entire cluster to be disabled and the applications will always have access to the data.

A highly-available cluster runs in a multi-master mode. A multi-master Kudu cluster has multiple masters, one of which is elected the leader by consensus of all the master nodes. To make sure leader election doesn't result in ties, clusters must have an odd number of masters. Leader masters are responsible for responding to client requests for data access. Masters that are not leaders are called followers. Followers keep their copy of the cluster metadata in sync with the leader, so that if a leader fails at any time, any of the followers can be elected by the remaining masters as the new leader.

High-availability is an optional feature that is useful when continuous access to the data is critical.

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 gray.



If the master fails when multi-master mode is disabled, tablet servers can continue to serve data as long as the required metadata from the master is cached. However, any requests to access data in tables for which the tablet servers don't have cached metadata will fail until a master is available again. Kudu masters are lightweight processes and can restart quickly.

A key feature to support fault tolerance is tablet replication, which ensures that data in a Kudu table will never be lost. By default, each tablet is replicated on three different servers. Each table can have its own setting for how many times its tablets should be replicated. Note that there must be enough tablet servers in the cluster to allow each replica to be stored on a separate server.

Among all of a tablet's replicas, one is elected leader by the tablet servers, the same way the cluster's Kudu masters elect a leader master. All other replicas are considered followers. Both followers and leaders can service reads, but only leaders can service a write operation.

The leader also propagates these operations to each of the follower replicas, who then perform the same operation with their own data, so that the follower and leader replicas are always in sync. In case a leader replica becomes unavailable, then the followers elect a new leader.

The default replication factor of three allows for continued operation if just one replica is unavailable. A replication factor of five can tolerate up to two unavailable replicas while still guaranteeing data availability.

Kudu-Impala integration

Apache Kudu has tight integration with Apache Impala, allowing you to use Impala to insert, query, update, and delete data from Kudu tablets using Impala's SQL syntax, as an alternative to using the Kudu APIs to build a custom

Kudu application. In addition, you can use JDBC or ODBC to connect existing or new applications written in any language, framework, or business intelligence tool to your Kudu data, using Impala as the broker.

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

Example use cases

Kudu gives you capability to stream inputs with near real-time availability. You can run time-series applications with varying access patterns, and develop predective learning models. Kudu gives you freedom to access and query data from any legacy sources or formats using Impala.

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.

Kudu concepts

In order to use Kudu efficientyl, you have to familiarize yourself with some basic concepts related to Kudu.

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 tablet 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:

Table 1: Contents of the catalog table

Content	Description
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 replications

Kudu replicates operations, not on-disk data. This is referred to as logical replication, as opposed to physical replication.

Logical replication 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.

Related Information

Raft consensus algorithm

Apache Kudu usage limitations

We recommend that you review the usage limitations of Kudu with respect to schema design, scaling, server management, cluster management, replication and backup, security, and integration with Spark and Impala.

Schema design limitations

Kudu currently has some known limitations that may factor into schema design.

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 tables can have a maximum of 300 columns. We recommend schema designs that use fewer columns for best performance.
- CHAR, and complex types such as ARRAY, MAP, and STRUCT are not supported.
- Type, nullability, and type attributes (i.e. precision and scale of DECIMAL, length of VARCHAR) of the existing columns cannot be changed by altering the table.
- Dropping a column does not immediately reclaim space. Compaction must run first.
- Kudu does not allow the type of a column to be altered after the table is created.

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

Other usage limitations

- Secondary indexes are not supported.
- Multi-row transactions are not supported.
- Relational features, such as foreign keys, are not supported.

- Identifiers such as column and table names are restricted to be valid UTF-8 strings. Additionally, a maximum length of 256 characters is enforced.

  If you are using Apache Impala to query Kudu tables, refer to the section on *Impala integration limitations* as well.
- Deleted row disk space cannot be reclaimed. The disk space occupied by a deleted row is only reclaimable via compaction, and only when the deletion's age exceeds the "tablet history maximum age" which is controlled by the --tablet_history_max_age_sec flag. Currently, Kudu only schedules compactions in order to improve read/write performance. A tablet will never be compacted purely to reclaim disk space. As such, range partitioning should be used when it is expected that large swaths of rows will be discarded. With range partitioning, individual partitions may be dropped to discard data and reclaim disk space. See KUDU-1625 for more details.

Partitioning limitations

Here are some of the limitations that you must consider before partitionoing tables.

- Tables must be manually pre-split into tablets using simple or compound primary keys. Automatic splitting is not yet possible. Kudu does not allow you to change how a table is partitioned after creation, with the exception of adding or dropping range partitions.
- Data in existing tables cannot currently be automatically repartitioned. As a workaround, create a new table with the new partitioning and insert the contents of the old table.
- Tablets that lose a majority of replicas (such as 1 left out of 3) require manual intervention to be repaired.

Scaling recommendations and limitations

Here are some of the recommended vales that you should consider while deciding the size of the tablet servers, masters, and other storage resources to get optimum performance.

Kudu can seamlessly run across a wide array of environments and workloads with minimal expertise and configuration at the following scale:

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

  The recommended target size for tablets is under 10 GiB.

Staying within these limits will provide the most predictable and straightforward Kudu experience. However, experienced users who run on modern hardware, use the latest versions of Kudu, test and tune Kudu for their use case, and work closely with the community, can achieve much higher scales comfortably. Following are some anecdotal values that have been seen in real world production clusters:

- Number of master servers: 3
- More than 300 tablet servers
- 10+ TiB of stored data per tablet server, post-replication and post-compression
- More than 4000 tablets per tablet server, post-replication
- 50 GiB of stored data per tablet. Going beyond this can cause issues such a reduced performance, compaction issues, and slower tablet startup time.

Server management limitations

Here are some of the server management guidelines that you should consider before implementing Kudu.

- Production deployments should configure a least 4 GiB of memory for tablet servers, and ideally more than 16 GiB when approaching the data and tablet scale limits.
- Write ahead logs (WALs) can only be stored on one disk.
- Data directories cannot be removed. You must reformat the data directories to remove them.
- 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

When managing Kudu clusters, review the following limitations and recommended maximum point-to-point latency and bandwidth values.

- Recommended maximum point-to-point latency within a Kudu cluster is 20 milliseconds.
- Recommended minimum point-to-point bandwidth within a Kudu cluster is 10 Gbps.
- If you intend to use the location awareness feature to place tablet servers in different locations, it is recommended that you measure the bandwidth and latency between servers to ensure they fit within the above guidelines.
- All masters must be started at the same time when the cluster is started for the very first time.

Impala integration limitations

Here are the limitations when integrating Kudu with Impala, and list of Impala keywords that are not supported for creating Kudu tables.

- 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.
- 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

Here are the limitations that you should consider while integrating Kudu and Spark.

- Spark 2.2 (and higher) requires Java 8 at runtime even though Kudu Spark 2.x integration is Java 7 compatible. Spark 2.2 is the default dependency version as of Kudu 1.5.0.
- Kudu tables with a name containing upper case or non-ASCII characters must be assigned an alternate name when registered as a temporary table.
- Kudu tables with a column name containing upper case or non-ASCII characters must not be used with SparkSQL. Columns can be renamed in Kudu to work around this issue.
- <> and ORpredicates 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 "FO O%BAR" won't.
- Kudu does not support all the types supported by Spark SQL. For example, Date and complex types are not supported.
- Kudu tables can only be registered as temporary tables in SparkSQL.
- Kudu tables cannot be queried using HiveContext.

Security limitations

Here are some limitations related to data encryption and authorization in Kudu.

- Data encryption at rest is not directly built into Kudu. Encryption of Kudu data at rest can be achieved through the use of local block device encryption software such as dmcrypt.
- Row-level authorization is not available.
- Kudu does not support configuring a custom service principal for Kudu processes. The principal must follow the pattern kudu/<HOST>@<DEFAULT.REALM>.
- Server certificates generated by Kudu IPKI are incompatible with bouncycastle version 1.52 and earlier.
- The highest supported version of the TLS protocol is TLSv1.2
- When you are creating a new Kudu service using the Ranger web UI, the Test Connection button is displayed. However, the TestConnection tab is not implemented in the Kudu Ranger plugin. As a result if you try to use it with Kudu it will fails, but that does not mean that the service is not working.

Other known issues

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

- If the Kudu master is configured with the -log_force_fsync_all option, the tablet servers and the 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 1000 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.

More resources for Apache Kudu

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

Resources

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.

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.