Cloudera Runtime 7.1.0

Apache Kudu Schema Design

Date published: 2020-02-28
Date modified:

Legal Notice

Contents

Apache Kudu schema design

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

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

The perfect schema

The perfect schema would accomplish the following:

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

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

Column design

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

Supported column types include:

- boolean
- 8-bit signed integer
- 16-bit signed integer
- 32-bit signed integer
- 64-bit signed integer
- unixtime_micros (64-bit microseconds since the Unix epoch)
- single-precision (32-bit) IEEE-754 floating-point number
- double-precision (64-bit) IEEE-754 floating-point number
- decimal
- UTF-8 encoded string (up to 64KB uncompressed)
- binary (up to 64KB uncompressed)
- VARCHAR type with configurable maximum length (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.

Attention: Unlike HBase, Kudu does not provide a version or timestamp column to track changes to a row. If version or timestamp information is needed, the schema should include an explicit version or timestamp column.

Decimal type

The decimal type is a numeric data type with fixed scale and precision suitable for financial and other arithmetic calculations where the imprecise representation and rounding behavior of float and double make those types impractical. The decimal type is also useful for integers larger than int64 and cases with fractional values in a primary key.

The decimal type is a parameterized type that takes precision and scale type attributes.

Precision represents the total number of digits that can be represented by the column, regardless of the location of the decimal point. This value must be between 1 and 38 and has no default. For example, a precision of 4 is required to represent integer values up to 9999, or to represent values up to 99.99 with two fractional digits. You can also represent corresponding negative values, without any change in the precision. For example, the range -9999 to 9999 still only requires a precision of 4.

Scale represents the number of fractional digits. This value must be between 0 and the precision. A scale of 0 produces integral values, with no fractional part. If precision and scale are equal, all of the digits come after the decimal point. For example, a decimal with precision and scale equal to 3 can represent values between -0.999 and 0.999.

Performance considerations:

Kudu stores each value in as few bytes as possible depending on the precision specified for the decimal column. For that reason it is not advised to just use the highest precision possible for convenience. Doing so could negatively impact performance, memory and storage.

Before encoding and compression:

- Decimal values with precision of 9 or less are stored in 4 bytes.
- Decimal values with precision of 10 through 18 are stored in 8 bytes.
- Decimal values with precision greater than 18 are stored in 16 bytes.

Note: The precision and scale of decimal columns cannot be changed by altering the table.

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

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

Prefix Encoding

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

Each column in a Kudu table can be created with an encoding, based on the type of the column. Starting with Kudu 1.3, default encodings are specific to each column type.

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

Column compression

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

By default, columns that are Bitshuffle-encoded are inherently compressed with the LZ4 compression. Otherwise, columns are stored uncompressed. Consider using compression if reducing storage space is more important than raw scan performance.

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

Primary key design

Every Kudu table must declare a primary key comprised of one or more columns. Like an RDBMS primary key, the Kudu primary key enforces a uniqueness constraint. Attempting to insert a row with the same primary key values as an existing row will result in a duplicate key error.

Primary key columns must be non-nullable, and may not be a boolean or floating-point type.

Once set during table creation, the set of columns in the primary key may not be altered.

Unlike an RDBMS, Kudu does not provide an auto-incrementing column feature, so the application must always provide the full primary key during insert.

Row delete and update operations must also specify the full primary key of the row to be changed. Kudu does not natively support range deletes or updates.

The primary key values of a column may not be updated after the row is inserted. However, the row may be deleted and re-inserted with the updated value.

Primary key index

As with many traditional relational databases, Kudu's primary key is in a clustered index. All rows within a tablet are sorted by its primary key.

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

⚠ **Attention:** Primary key indexing optimizations apply to scans on individual tablets. See the Partition pruning on page 8 topic for details on how scans can use predicates to skip entire tablets.

Considerations for backfill inserts

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

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

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

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

- Make the primary keys more compressible.

  For example, with the first column of a primary key being a random ID of 32-bytes, caching one billion primary keys would require at least 32 GB of RAM to stay in cache. If caching backfill primary keys from several days ago, you need to have several times 32 GB of memory. By changing the primary key to be more compressible, you increase the likelihood that the primary keys can fit in cache and thus reducing the amount of random disk I/Os.
- Use SSDs for storage as random seeks are orders of magnitude faster than spinning disks.
- Change the primary key structure such that the backfill writes hit a continuous range of primary keys.

Partitioning

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

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

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

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

Range partitioning

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

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

Adding and Removing Range Partitions

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

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

Hash partitioning

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

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

Multilevel partitioning

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

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

Partition pruning

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

Partitioning examples

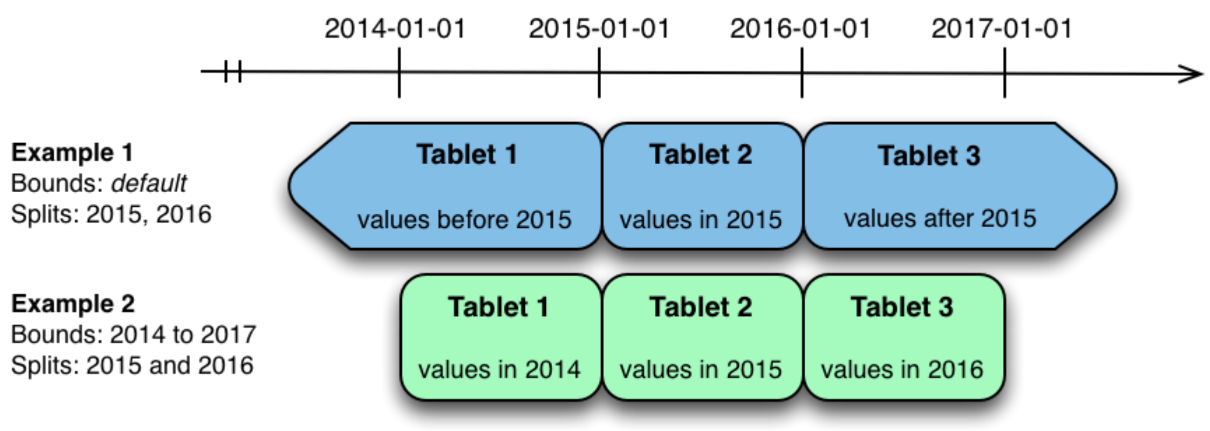
To illustrate the factors and tradeoffs associated with designing a partitioning strategy for a table, we will walk through some different partitioning scenarios.

Consider the following table schema for storing machine metrics data (using SQL syntax and date-formatted timestamps for clarity):

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

Range partitioning

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



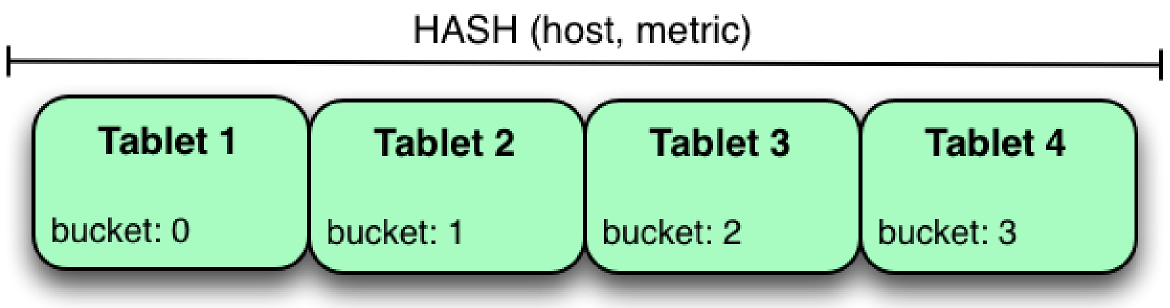
The image above shows the two ways the metrics table can be range partitioned on the time column. In the first example (in blue), the default range partition bounds are used, with splits at 2015-01-01 and 2016-01-01. This results in three tablets: the first containing values before 2015, the second containing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

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

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

Hash partitioning

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



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

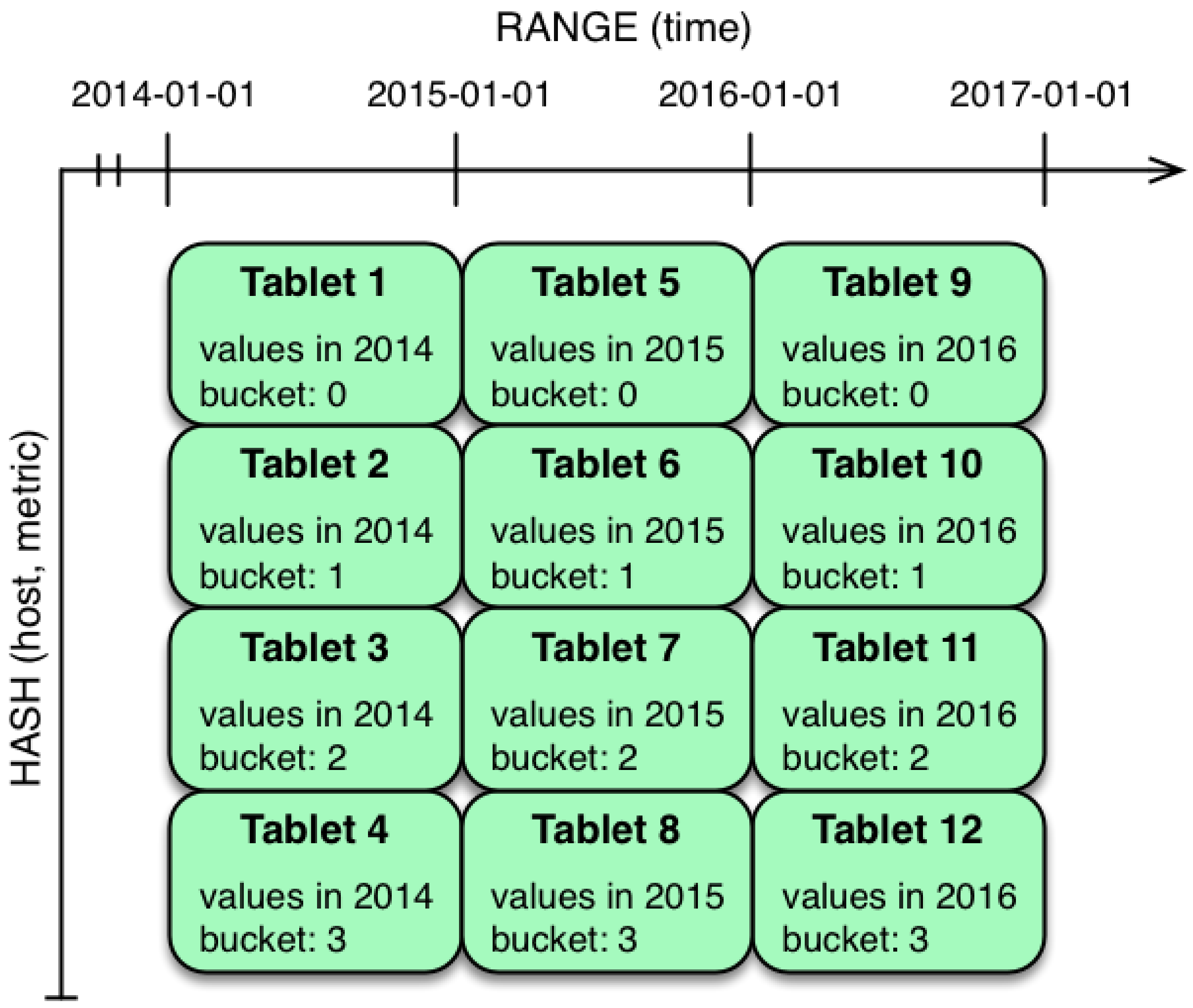
Hash and range partitioning

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

Table 1: 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

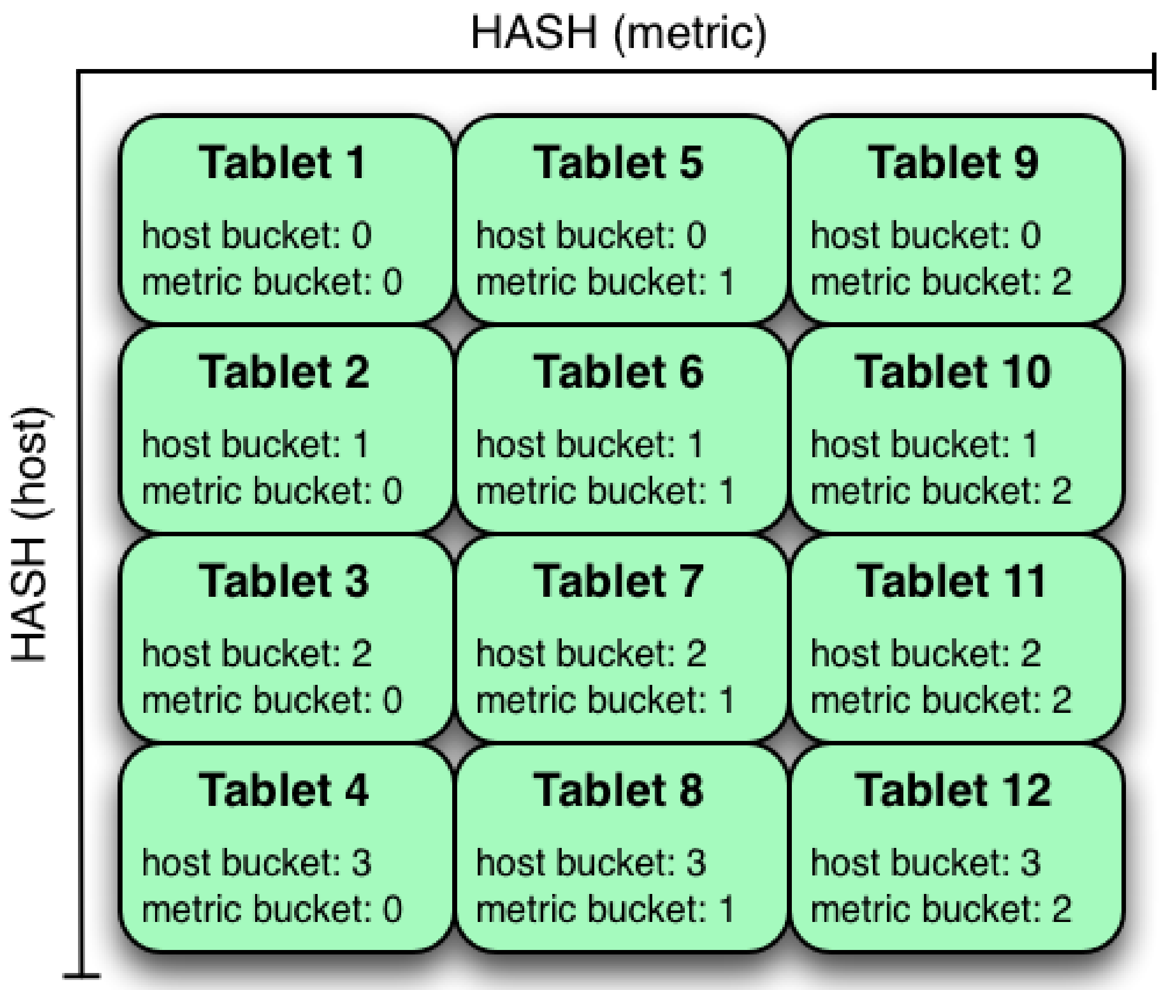
Hash partitioning is good at maximizing write throughput, while range partitioning avoids issues of unbounded tablet growth. Both strategies can take advantage of partition pruning to optimize scans in different scenarios. Using multilevel partitioning, it is possible to combine the two strategies in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


RANGE (time)
2014-01-01
2015-01-01
2016-01-01
2017-01-01
HASH (host, metric)
Tablet 1 values in 2014 bucket: 0
Tablet 5 values in 2015 bucket: 0
Tablet 9 values in 2016 bucket: 0
Tablet 2 values in 2014 bucket: 1
Tablet 6 values in 2015 bucket: 1
Tablet 10 values in 2016 bucket: 1
Tablet 3 values in 2014 bucket: 2
Tablet 7 values in 2015 bucket: 2
Tablet 11 values in 2016 bucket: 2
Tablet 4 values in 2014 bucket: 3
Tablet 8 values in 2015 bucket: 3
Tablet 12 values in 2016 bucket: 3

In the example above, range partitioning on the time column is combined with hash partitioning on the host and metr ic 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

This topic lists the ways in which you can alter a table's schema.

- 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.

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, DATE, 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.

Apache Kudu transaction semantics

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

Kudu currently allows the following operations:

- Scans are read operations that can traverse multiple tablets and read information with different levels of 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.

Related Information

Spanner

Calvin

Technical Report: HybridTime - Accessible Global Consistency with High Clock Uncertainty

Single tablet write operations

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

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

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

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

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

Writing to multiple tablets

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

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

- CLIENT_PROPAGATED Consistency

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

  Java Client

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

  C++ Client

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

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

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

The COMMIT_WAIT consistency mode may be selected as follows:

Java Client

Call KuduSession#setExternalConsistencyMode(ExternalConsistencyMode.COMMIT_WAIT)

C++ Client

Call KuduSession::SetExternalConsistencyMode(COMMIT_WAIT)

Warning:

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

Read operations (scans)

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

The mode may be selected as follows:

Java Client

Call KuduScannerBuilder#ReadMode(…)

C++ Client

Call KuduScanner::SetReadMode()

The following modes are available in both clients:

READ_LATEST

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

READ_AT_SNAPSHOT

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

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

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

Note:

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

Java Client

Call KuduScannerBuilder#replicaSelection(...)

C++ Client

Call KuduScanner::SetSelection(...)

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

Related Information

Known issues and limitations

Known issues and limitations

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

Writes

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

Recommendation

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

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

Reads (scans)

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

Recommendation

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

Impala scans are currently performed as READ_LATEST and have no consistency guarantees.

In AUTO_BACKGROUND_FLUSH mode, or when using "async" flushing mechanisms, writes applied to a single client session may get reordered due to the concurrency of flushing the data to the server. This is particularly noticeable if a single row is quickly updated with different values in succession. This phenomenon affects all client API implementations. Workarounds are described in the respective API documentation for FlushMode or AsyncKuduSession. See KUDU-1767.

Scaling Kudu

This section describes in detail how Kudu scales with respect to various system resources, including memory, file descriptors, and threads. See Scaling recommendations and limitations for the maximum recommended parameters of a Kudu cluster. They can be used to estimate roughly the number of servers required for a given quantity of data.

Attention: The recommendations and conclusions here are only approximations. Appropriate numbers depend on use case. There is no substitute for measurement and monitoring of resources used during a representative workload.

Terms

Here are some of the terms used in this topic.

- hot replica: A tablet replica that is continuously receiving writes. For example, in a time series use case, tablet replicas for the most recent range partition on a time column would be continuously receiving the latest data, and would be hot replicas.
- cold replica: A tablet replica that is not hot, i.e. a replica that is not frequently receiving writes, for example, once every few minutes. A cold replica may be read from. For example, in a time series use case, tablet replicas for previous range partitions on a time column would not receive writes at all, or only occasionally receive late updates or additions, but may be constantly read.
- data on disk: The total amount of data stored on a tablet server across all disks, post-replication, post-compression, and post-encoding.

Example workload

The sections below help you perform sample calculations using the following parameters:

- 200 hot replicas per tablet server
- 1600 cold replicas per tablet server
- 8TB of data on disk per tablet server (about 4.5GB/replica)
- 512MB block cache
- 40 cores per server
- limit of 32000 file descriptors per server
- a read workload with 1 frequently-scanned table with 40 columns

This workload resembles a time series use case, where the hot replicas correspond to the most recent range partition on time.

Memory

The flag --memory_limit_hard_bytes determines the maximum amount of memory that a Kudu tablet server may use. The amount of memory used by a tablet server scales with data size, write workload, and read concurrency.

The following table provides numbers that can be used to compute a rough estimate of memory usage:

Table 2: Tablet server memory usage

Type	Multiplier	Description
Memory required per TB of data on disk	1.5GB per 1TB data on disk	Amount of memory per unit of data on disk required for basic operation of the tablet server.

Type	Multiplier	Description
Hot Replicas' MemRowSets and DeltaMemStores	minimum 128MB per hot replica	Minimum amount of data to flush per MemRowSet flush. For most use cases, updates should be rare compared to inserts, so the DeltaMemStores should be very small.
Scans	256KB per column per core for read-heavy tables	Amount of memory used by scanners, and which will be constantly needed for tables which are constantly read.
Block Cache	Fixed by --block_cache_capacity_mb(default 512MB)	Amount of memory reserved for use by the block cache.

Using this information for the example load gives the following breakdown of memory usage:

Table 3: Example tablet server memory usage

Type	Amount
8TB data on disk	8TB * 1.5GB / 1TB = 12GB
200 hot replicas	200 * 128MB = 25.6GB
1 40-column, frequently-scanned table	40 * 40 * 256KB = 409.6MB
Block Cache	--block_cache_capacity_mb=512 = 512MB
Expected memory usage	38.5GB
Recommended hard limit	52GB

Using this as a rough estimate of Kudu's memory usage, select a memory limit so that the expected memory usage of Kudu is around 50-75% of the hard limit.

Verifying if a memory limit is sufficient

After configuring an appropriate memory limit with --memory_limit_hard_bytes, run a workload and monitor the Kudu tablet server process's RAM usage. The memory usage should stay around 50-75% of the hard limit, with occasional spikes above 75% but below 100%. If the tablet server runs above 75% consistently, the memory limit should be increased.

Additionally, it's also useful to monitor the logs for memory rejections, which look like:

```
Service unavailable: Soft memory limit exceeded (at 96.35% of capacity)
```

And watch the memory rejections metrics:

- leader_memory_pressure_rejections
- follower_memory_pressure_rejections
- transaction_memory_pressure_rejections

Occasional rejections due to memory pressure are fine and act as backpressure to clients. Clients will transparently retry operations. However, no operations should time out.

File descriptors

Processes are allotted a maximum number of open file descriptors (also referred to as fds). If a tablet server attempts to open too many fds, it will typically crash with a message saying something like "too many open files".

The following table summarizes the sources of file descriptor usage in a Kudu tablet server process:

Table 4: Tablet server file descriptor usage

Type	Multiplier	Description
File cache	Fixed by --block_manager_max_open_files (default 40% of process maximum)	Maximum allowed open fds reserved for use by the file cache.
Hot replicas	2 per WAL segment, 1 per WAL index	Number of fds used by hot replicas. See below for more explanation.
Cold replicas	3 per cold replica	Number of fds used per cold replica: 2 for the single WAL segment and 1 for the single WAL index.

Every replica has at least one WAL segment and at least one WAL index, and should have the same number of segments and indices; however, the number of segments and indices can be greater for a replica if one of its peer replicas is falling behind. WAL segment and index fds are closed as WALs are garbage collected.

Using this information for the example load gives the following breakdown of file descriptor usage, under the assumption that some replicas are lagging and using 10 WAL segments:

Table 5: Example tablet server file descriptor usage

Type	Amount
file cache	40% * 32000 fds = 12800 fds
1600 cold replicas	1600 cold replicas * 3 fds / cold replica = 4800 fds
200 hot replicas	(2 / segment * 10 segments/hot replica * 200 hot replicas) + (1 / index * 10 indices / hot replica * 200 hot replicas) = 6000 fds
Total	23600 fds

So for this example, the tablet server process has about 32000 - 23600 = 8400 fds to spare.

There is typically no downside to configuring a higher file descriptor limit if approaching the currently configured limit.

Threads

Processes are allotted a maximum number of threads by the operating system, and this limit is typically difficult or impossible to change. Therefore, this section is more informational than advisory.

If a Kudu tablet server’s thread count exceeds the OS limit, it will crash, usually with a message in the logs like "pthread_create failed: Resource temporarily unavailable". If the system thread count limit is exceeded, other processes on the same node may also crash.

Threads and thread pools are used all over Kudu for various purposes, but the number of threads found in nearly all of these does not scale with load or data/tablet size; instead, the number of threads is either a hard coded constant, a constant defined by a configuration parameter, or based on a static dimension (such as the number of CPU cores).

The only exception to this is the WAL append thread, one of which exists for every "hot" replica.

Note that all replicas may be considered hot at startup, so tablet servers' thread usage will generally peak when started and settle down thereafter.