Cloudera Runtime 7.3.2

# Managing Apache Kudu

**Date published: 2020-07-28**
**Date modified: 2026-03-31**

**https://docs.cloudera.com/**

# Legal Notice

# Contents

# Limitations

Review the server management and cluster management guidelines that you should consider before implementing Kudu.

## 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 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.

## 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.

# Start and stop Kudu processes

You can start, stop, and configure Kudu services to start automatilcally by using the CLI commands.

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
```

# Orchestrate a rolling restart with no downtime

Kudu 1.12 provides tooling to restart a cluster with no downtime. This topic provides the steps to perform rolling restart.

## About this task

**Note:** If any tables in the cluster have a replication factor of 1, some quiescing tablet servers will never become fully quiesced, as single-replica tablets do not naturally relinquish leadership. If such tables exist, use the kudu cluster rebalance tool to move replicas of these tables away from the quiescing tablet server by specifying the --ignored_tservers, --move_replicas_from_ignored_tservers, and --tables options.

**Note:** If running with rack awareness, the following steps can be performed by restarting multiple tablet servers within a single rack at the same time. Use ksck to ensure that the location assignment policy is enforced while going through these steps, and that no more than a single location is restarted at the same time. At least three locations should be defined in the cluster to safely restart multiple tablet service within one location.

Cloudera Manager can automate this process, by using the “Rolling Restart” command on the Kudu service.

**Note:** Cloudera Manager does not support automatic moving of the single-replica tablets.

Cloudera Manager will prompt you to specify how many tablet servers to restart concurrently. If running with rack awareness with and at least three racks specified across all hosts that contain Kudu roles, it is safe to specify the restart batch with up to one rack at a time, provided the rack assignment policy is being enforced.

The following service configurations can be set to tune the parameters the rolling restart will run with:

- Rolling Restart Health Check Interval: the interval in seconds that Cloudera Manager will run ksck after restarting a batch of tablet servers, waiting for the cluster to become healthy.
- Maximum Allowed Runtime to Rolling Restart a Batch of Servers: the total amount of time in secondsCloudera Manager will wait for the cluster to become healthy after restarting a batch of tablet servers, before exiting with an error.

## Procedure

1. Restart the master(s) one-by-one. If there is only a single master, this may cause brief interference with on-going workloads.
2. Starting with a single tablet server, put the tablet server into maintenance mode by using the kudu tserver state e nter_maintenance tool.
3. Start quiescing the tablet server using the kudu tserver quiesce start tool. This signals Kudu to stop hosting leaders on the specified tablet server and to redirect new scan requests to other tablet servers.
4. Periodically run kudu tserver quiesce start with the --error_if_not_fully_quiesced option, until it returns success, indicating that all leaders have been moved away from the tablet server and that all on-going scans have completed.
5. Restart the tablet server.
6. Periodically run ksck until the cluster ireports a healthy status.
7. Exit maintenance mode on the tablet server by running kudu tserver state exit_maintenance. This allows new tablet replicas to be placed on the tablet server.
8. Repeat these steps for all tablet servers in the cluster.

## Related Information

Changing directory configuration

## Minimize cluster disruption during temporary planned downtime of a single tablet server

If a single tablet server is brought down temporarily in a healthy cluster, all tablets will remain available and clients will function as normal, after potential short delays due to leader elections. However, if the downtime lasts for more than --follower_unavailable_considered_failed_sec (default 300) seconds, the tablet replicas on the down tablet server will be replaced by new replicas on available tablet servers. This will cause stress on the cluster as tablets re-replicate and, if the downtime lasts long enough, significant reduction in the number of replicas on the down tablet server, which may require the rebalancer to fix.

To work around this, in Kudu versions 1.11 onward, the kudu CLI contains a tool to put tablet servers into maintenance mode. While in this state, the tablet server's replicas are not re-replicated due to its downtime alone, though re-replication may still occur in the event that the server in maintenance suffers from a disk failure or if a follower replica on the tablet server falls too far behind its leader replica. Upon exiting maintenance, re-replication is triggered for any remaining under-replicated tablets.

The kudu tserver state enter_maintenance and kudu tserver state exit_maintenance tools are added to orchestrate tablet server maintenance. The following can be run from a tablet server to put it into maintenance:

```
$ TS_UUID=$(sudo -u kudu kudu fs dump uuid --fs_wal_dir=<wal_dir> --fs_data_
dirs=<data_dirs>)
$ sudo -u kudu kudu tserver state enter_maintenance <master_addresses> "$T
S_UUID"
```

The tablet server maintenance mode is shown in the "Tablet Servers" page of the Kudu leader master's web UI, and in the output of kudu cluster ksck. To exit maintenance mode, run the following command:

```
sudo -u kudu kudu tserver state exit_maintenance <master_addresses> "$TS_UUI
D"
```

# 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.

# Best practices when adding new tablet servers

A common workflow when administering a Kudu cluster is adding additional tablet server instances, in an effort to increase storage capacity, decrease load or utilization on individual hosts, increase compute power, and more.

By default, any newly added tablet servers will not be utilized immediately after their addition to the cluster. Instead, newly added tablet servers will only be utilized when new tablets are created or when existing tablets need to be replicated, which can lead to imbalanced nodes. It's recommended to run the rebalancer CLI tool just after adding a new tablet server into the cluster.

Avoid placing multiple tablet servers on a single node. Doing so nullifies the point of increasing the overall storage capacity of a Kudu cluster and increases the likelihood of tablet unavailability when a single node fails (the latter drawback is not applicable if the cluster is properly configured to use the rack awareness (location awareness) feature.

To add additional tablet servers to an existing cluster, the following steps can be taken to ensure tablet replicas are uniformly distributed across the cluster:

1. Ensure that Kudu is installed on the new machines being added to the cluster, and that the new instances have been correctly configured to point to the pre-existing cluster. Then, start the new tablet server instances.
2. Verify that the new instances check in with the Kudu Master(s) successfully. A quick method for verifying whether they have successfully checked in with the existing Master instances is to view the Kudu Master WebUI, specifically the /tablet-servers section, and validate that the newly added instances are registered, and have a heartbeat.
3. Once the tablet server(s) are successfully online and healthy, follow the steps to run the rebalancing tool which spreads the existing tablet replicas to the newly added tablet servers.
4. After the rebalancer tool has completed, or even during its execution, you can check the health of the cluster using the ksck command-line utility.

# Decommission or remove a tablet server

You can decommission or permanently remove a tablet server from a cluster.

## About this task

Starting with Kudu 1.12, the Kudu rebalancer tool can be used to decommission a tablet server by supplying the --ignored_tservers and --move_replicas_from_ignored_tservers arguments.

**Note:** Do not decommission multiple tablet servers at once. To remove multiple tablet servers from the cluster, follow the below instructions for each tablet server, ensuring that the previous tablet server is removed from the cluster and ksck is healthy before shutting down the next.

## Procedure

1. Ensure the cluster is in good health using ksck.
2. Put the tablet server into a maintenance mode by using the kudu tserver state enter_maintenance tool.

3. Run the kudu cluster rebalance tool, supplying the --ignored_tservers argument with the UUIDs of the tablet servers to be decommissioned, and the --move_replicas_from_ignored_tservers flag.
4. Wait for the moves to complete and for ksck to show the cluster in a healthy state.

   **Note:** To verify that all the moves are completed from the server placed in maintenance mode, ensure that there are no replicas on the server by running kudu cluster ksck or running kudu cluster rebalancer with --report_only and --output_replica_distribution_details flags or checking the Web UI of the tablet server.
5. The decommissioned tablet server can be brought offline by stopping the tablet server in Cloudera Manager.
6. To completely remove it from the cluster so ksck shows the cluster as completely healthy, restart the masters.

   If you have only one master in your deployment, this may cause cluster downtime. In a multi-master deployment, restart the masters in sequence to avoid cluster downtime.

### Related Information

Minimize cluster disruption during temporary planned downtime of a single tablet server

# Use cluster names in the kudu command line tool

When using the kudu command line tool, it can be difficult to remember the precise list of Kudu master RPC addresses needed to communicate with a cluster, especially when managing multiple clusters. As an alternative, you can use the command line tool to identify clusters by name.

### Procedure

1. Create a new directory to store the Kudu configuration file.
2. Export the path to this newly created directory in the KUDU_CONFIG environment variable.
3. Create a file called kudurc in the new directory.
4. Populate kudurc as follows, substituting your own cluster names and RPC addresses:

```
clusters_info:
  cluster_name1:
    master_addresses: ip1:port1,ip2:port2,ip3:port3
  cluster_name2:
    master_addresses: ip4:port4
```

5. When using the kudu command line tool, replace the list of Kudu master RPC addresses with the cluster name, prepended with the character @. For example:

```
$ sudo -u kudu kudu cluster @cluster_name1
```

**Note:** Cluster names may be used as input in any invocation of the kudu command line tool that expects a list of Kudu master RPC addresses.

# Migrate Kudu data from one directory to another on the same host

Take the following steps to move the entire Kudu data from one directory to another.

### About this task

**Note:** The steps were verified on an environment where the master and the server instances were configured to write the WAL/Data to the same directory.

### Procedure

1. Stop the Kudu service.
2. Modify the directory configurations for the Master/Server instances.
3. Move the existing data from the old directory, to the new one.
4. Make sure the file/directory ownership is set to the kudu user.
5. Restart the Kudu service.
6. Run ksck and verify for the healthy status.

### Related Information

Changing directory configuration

# Migrate to a multiple Kudu master configuration

Before migrating to a multiple Kudu masters set up, you need to perform many migration planning steps, such as deciding the number of masters, and choosing the nodes where to add the new Kudu masters.

### About this task

The migration procedure does not require stopping all the Kudu processes. The restarting of the Kudu processes can be done without incurring downtime.

**Important:** The procedure does not support adding multiple masters at the same time by selecting multiple hosts. It is only supported to add one by one.

### Before you begin

- You must decide how many masters to use.

  The number of masters should be odd because an even number of masters does not provide any benefit over having one fewer masters. Three or five node master configurations are recommended as they can tolerate the failure of one or two masters respectively.
- You must establish a maintenance window.

  One hour should be sufficient maintenance window time. During this time the Kudu cluster might become unavailable if there is some problem during the procedure.

### Procedure

1. Configure a DNS alias for the new master.

   The alias can be:

   - a CNAME record: if the machine already has an A record in DNS
   - and A record: if the machine is only known by its IP address

   **Important:** Without DNS aliases, it is not possible to recover from permanent master failures without restarting the masters and tablet servers in the cluster to pick up the replacement master node with a different hostname. It is highly recommended that you use DNS aliases.
2. Perform the following preparatory steps for the new master that you are planning to add:
   a) Choose a node in the cluster where there is no running Kudu master yet and which has enough spare CPU and memory capacity to host a new Kudu master.

      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.
   b) Choose and record the directories where the new master's data and WAL will be located.
   c) Choose and record the port the master should use for RPCs.

3. In Cloudera Manager, add a new Kudu Master role to the selected new master node:
   a) Select the Kudu service.
   b) Select Instances.
   c) Click Add Role Instances.
   d) Click Select hosts under Master × 1.
   e) Select the host node and click OK.
   f) Click Continue.
   g) Review the changes and if everything is correct, click Finish.

   Now, the newly added master role instance is commissioned but not part of the cluster. To make it a part of the cluster, you need to start it.
4. Start the newly added master role instance.
   a) Select the master role instance.
   b) Go to Actions for Selected Start .

   Upon starting, the new Kudu master registers with the existing master(s).
5. Review the log files to make sure no errors have been reported, and click Close.

   Now, the newly added master role instance is a part of the cluster.
6. To propagate the new master membership configuration, restart all Kudu masters (the old ones and the newly added one).

   It can be done on a one-by-one (i.e. Rolling Restart) or all-at-once manner.
7. After all the Kudu masters are up and running, open the Web UI of the newly added master and click the Masters tab.

   A page with **Live Masters** table appears.
8. Verify the following:
   - The table shows all the Kudu masters; the old ones and the newly added one.
   - There is one master marked as LEADER and the rest are marked as FOLLOWER in the Role column.
9. In the Live Masters table, click the UUID link of the leader Kudu master in the UUID column of the corresponding row.

   The Web UI of the current leader master appears.
10. In the Web UI of the current leader master, click the Masters tab.

    A page with Live Masters table appears.
11. Verify the following:
    - The table shows all the Kudu masters in the cluster; the old ones and the newly added one.
    - The information in the Registration column makes sense for the newly added Kudu master: the entries in the rpc_addresses field RPC match the hostnames and/or IP addresses of the nodes hosting KUDU_MASTER roles.
    - The information in the Start time column reflects the time of the recent restart performed at step 6.

**Note:** Now, the newly added Kudu master role instance is a part of the cluster. To add one more Kudu master into the cluster, repeat steps 1-11.

## What to do next

1. If you have Kudu tables that are accessed from Impala and you did not set up DNS aliases, update the HMS database manually in the underlying database that provides the storage for HMS:
   **a.** Connect to the HMS database.
   **b.** Run an SQL statement similar to the following example:

```
UPDATE TABLE_PARAMS
SET PARAM_VALUE =
```

```
  'master-1.example.com,master-2.example.com,master-3.example.com'
WHERE PARAM_KEY = 'kudu.master_addresses' AND PARAM_VALUE = 'master-1.e
xample.com';
```

c. In impala-shell run the following command: INVALIDATE METADATA;

2. After adding all the desired masters into the cluster, 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>, where
   - hostname is master's hostname
   - port is master's RPC port number
3. Restart all the tablet servers to pick up the new masters' configuration.
4. To verify that all masters are working properly, consider performing the following checks:
   - Using a browser, visit each master's web UI and navigate to the /masters page.

     All the masters should 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.
   - If applicable, run a few quick SQL queries against a couple of migrated Kudu tables using impala-shell or Data Explorer.

# Change master hostnames

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

## Prepare for master hostname changes

In this step, you need to identify a down-time window, and note the UUID and the RPC address of each master.

### Procedure

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

## Perform master hostname changes

You need to bring the Kudu clusters down to update the master hostnames. Therefore, identify at least a one-hour maintenance window for this task.

## Procedure

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

```
$ sudo -u kudu kudu local_replica cmeta rewrite_raft_config --fs_wal_dir
=<master_wal_dir> [--fs_data_dirs=<master_data_dir>] 000000000000000000
0000000000000 <all_masters>
```

   For example:

```
$ sudo -u kudu kudu local_replica cmeta rewrite_raft_config --fs_wal_dir=/
data/kudu/master/wal --fs_data_dirs=/data/kudu/master/data 000000000000000
00000000000000000 4aab798a69e94fab8d77069edff28ce0:new-master-name-1:705
1 f5624e05f40649b79a757629a69d061e:new-master-name-2:7051 988d8ac6530f42
6cbe180be5ba52033d:new-master-name-3:7051
```

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

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

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

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

   b) In impala-shell, run:

```
INVALIDATE METADATA;
```

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

## Related Information

Monitoring cluster health with ksck

# Removing Kudu masters through Cloudera Manager

In the event that a multi-master deployment has overallocated nodes, you can remove or decommission the unwanted masters through Cloudera Manager.

### About this task

You can not decommission a leader master role from the cluster. You can only decommission a follower master role. If you want to decommission multiple master roles, do it one by one.

**Note:**

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

### Before you begin

You must have multiple master roles added in your cluster.

### Procedure

1. In Cloudera Manager, go to Kudu Instances .
2. Select the master role instance you want to decommission, and click Actions for Selected Decommission .
3. Review the changes in log files, and click Close.

   Now, the master role instance is decommissioned from the cluster.

# Recommissioning Kudu masters through Cloudera Manager

You can recommission any decommissioned master role instance in a multi-master deployment and include the instance in the cluster.

### Before you begin

You must have decommissioned a master role instance.

### Procedure

1. In Cloudera Manager, go to Kudu Instances .
2. Select the master role instance you want to recommission, and click Actions for Selected Recommission .
3. Review the changes in log files, and click Close.

   Now, the recommissioned master role instance is commissioned but is not part of the cluster. To make it a part of the cluster, you need to start it.
4. Start the newly added master role instance or instances.
   a) Select the master role instance.
   b) Click Actions for Selected Start .
5. Review the changes in log files, and click Close.

   Now, the recommissioned master role instance is a part of the cluster.

# Remove Kudu masters through CLI

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

**Important:**

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

## Prepare for removal

In order to remove the unwanted masters from a multi-master deployment, you need to identify them and note their UUID and RPC addresses.

### Procedure

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

## Perform the removal

When you remove any Kudu masters from a multi-master deployment, you need to rewrite the Raft configuration on the remaining masters, remove data and WAL directories from the unwanted masters, and finaly modify the value of the tserver_master_addrs configuration parameter for the tablet servers to remove the unwanted masters. You need to bring the Kudu clusters down. Therefore, identify at least a one-hour maintenance window for this task.

### Procedure

1. Perform the Raft configuration change by running the kudu master remove tool.

   Only a single master can be removed at a time. If multiple masters need to be removed, run the tool multiple times. In the following example master-2 is being removed from a Kudu cluster with two masters master-1, and master-2:

   ```
   $ sudo -u kudu kudu master remove master-1,master-2 master-2
   ```

2. Remove the data directories and WAL directory on the unwanted masters. This is a precaution to ensure that they cannot start up again and interfere with the new multi-master deployment.
3. Modify the value of the master_addresses configuration parameter for the masters of the new multi-master deployment.
4. Restart all of the masters that were not removed.
5. If you are not using Cloudera Manager: Modify the value of the tserver_master_addrs configuration parameter for the tablet servers to remove any unwanted masters.
6. Retart all of the tablet servers.

### What to do next

To verify that all masters are working properly, consider performing the following 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.

### Related Information

Configure Kudu processes

Monitoring cluster health with ksck

# How Range-aware replica placement in Kudu works

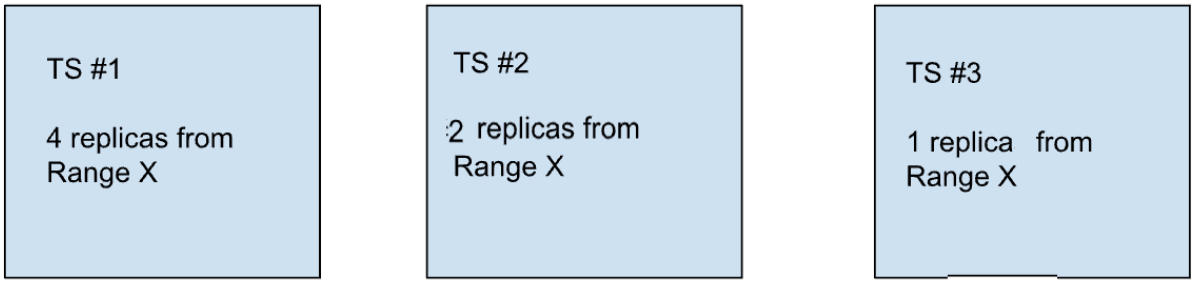
Learn how the range-aware replica placement algorithm works during initial tablet replica placement in Kudu tablet servers.

Kudu places new tablet replicas using an algorithm which is both range and table aware. This algorithm helps to avoid hotspotting that occurs if many replicas from the same range are placed on the same few tablet servers. Hotspotting causes tablet servers to be overwhelmed with write or read requests and can result in increased latency for these requests. To avoid hotspotting, this algorithm avoids targeting the same set of tablet servers for a set of replicas created in parallel. Rather, it spreads the replicas across multiple tablet servers.

This algorithm is independent of the location. Replica placement can be broken down into two steps; location selection and tablet server placement.

The range-aware algorithm works in the following way:

1. It ranks the tablet servers (TS) according to number of replicas per corresponding range (Range X in the following image).




2. It places the replica on the tablet server with the least amount of replicas from the corresponding range.

If two tablet servers have the same number of replicas from a range, the number of replicas from the table the range belongs to, is used as a tiebreaker. If the tablet servers contain the same number of replicas from the table as well, the number of total replicas hosted by the tablet servers is used as a final tiebreaker.

When determining the number of replicas of a range or table per tablet server, the algorithm uses two separate metrics. The first metric is the number of existing live replicas from the range or table. The second metric is the number of pending replicas from the range or table being placed. This metric is incremented when a pending replica is placed, but eventually the value decays to zero.

Replicas by range are stored with the table the range belongs to, in case multiple tables have ranges with the same range start keys. If the table is not stored along with the range, the algorithm can not differentiate between two ranges with the same start keys but on different tables.

**Important:** Both the range start key and the table ID for a certain range must be defined for this feature to work on that range.

The power of two choices algorithm, used earlier, was quick and efficient but did not prevent hotspotting.

# Run the tablet rebalancing tool

The kudu CLI contains a rebalancing tool that can be used to rebalance tablet replicas among tablet servers. For each table, the tool attempts to balance the number of replicas per tablet server. It also, without unbalancing any table, attempts to even out the number of replicas per tablet server across the cluster as a whole.

The rebalancing tool should be run as the Kudu admin user, specifying all master addresses:

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

When run, the rebalancer reports on the initial tablet replica distribution in the cluster, logs the replicas it moves, and prints a final summary of the distribution when it terminates:

```
Per-server replica distribution summary:
       Statistic       |   Value
-----------------------+-----------
 Minimum Replica Count | 0
 Maximum Replica Count | 24
 Average Replica Count | 14.400000
Per-table replica distribution summary:
 Replica Skew |  Value
--------------+----------
 Minimum      | 8
 Maximum      | 8
 Average      | 8.000000

I0613 14:18:49.905897 3002065792 rebalancer.cc:779] tablet e7ee9ade95b342a7a
94649b7862b345d: 206a51de1486402bbb214b5ce97a633c -> 3b4d9266ac8c45ff9a5d4d7
c3e1cb326 move scheduled
I0613 14:18:49.917578 3002065792 rebalancer.cc:779] tablet 5f03944529f44626
a0d6ec8b1edc566e: 6e64c4165b864cbab0e67ccd82091d60 -> ba8c22ab030346b4baa289
d6d11d0809 move scheduled
I0613 14:18:49.928683 3002065792 rebalancer.cc:779] tablet 9373fee3bfe74ce
c9054737371a3b15d: fab382adf72c480984c6cc868fdd5f0e -> 3b4d9266ac8c45ff9a5d4
d7c3e1cb326 move scheduled

... (full output elided)

I0613 14:19:01.162802 3002065792 rebalancer.cc:842] tablet f4c046f18b174cc
2974c65ac0bf52767: 206a51de1486402bbb214b5ce97a633c -> 3b4d9266ac8c45ff9a5d4
d7c3e1cb326 move completed: OK

rebalancing is complete: cluster is balanced (moved 28 replicas)
Per-server replica distribution summary:
       Statistic       |   Value
-----------------------+-----------
 Minimum Replica Count | 14
 Maximum Replica Count | 15
 Average Replica Count | 14.400000

Per-table replica distribution summary:
 Replica Skew |  Value
--------------+----------
 Minimum      | 1
 Maximum      | 1
 Average      | 1.000000
```

If more details are needed in addition to the replica distribution summary, use the --output_replica_distribution_details flag. If added, the flag makes the tool print per-table and per-tablet server replica distribution statistics as well.

Use the --report_only flag to get a report on table-wide and cluster-wide replica distribution statistics without starting any rebalancing activity.

The rebalancer can also be restricted to run on a subset of the tables by supplying the --tables flag. Note that, when running on a subset of tables, the tool does not attempt to balance the cluster as a whole.

The length of time rebalancing is run for can be controlled with the flag --max_run_time_sec. By default, the rebalancer runs until the cluster is balanced. To control the amount of resources devoted to rebalancing, modify the flag --max_moves_per_server. See kudu cluster rebalance --help for more.

It is safe to stop the rebalancer tool at any time. When restarted, the rebalancer continues rebalancing the cluster.

The rebalancer tool requires all registered tablet servers to be up and running to proceed with the rebalancing process in order to avoid possible conflicts and races with the automatic re-replication and to keep replica placement optimal for current configuration of the cluster. If a tablet server becomes unavailable during the rebalancing session, the rebalancer exits. As noted above, it is safe to restart the rebalancer after resolving the issue with unavailable tablet servers. However, if it is necessary to rebalance the cluster when a few tablet servers in a Kudu cluster are not available, it is possible to specify their UUIDs as a comma-separated list with the --\ignored_tservers flag and rebalance the rest of the cluster. With the --ignored_tservers flag, the specified tablet servers are effectively ignored by the rebalancer tool, which means they are not considered as a part of the cluster along with tablet replicas they host.

The rebalancing tool can rebalance Kudu clusters running older versions as well, with some restrictions. Consult the following table for more information. In the table, "RF" stands for "replication factor".

| Version Range | Rebalances RF = 1 Tables? | Rebalances RF > 1 Tables? |
|---|---|---|
| v < 1.4.0 | No | No |
| 1.4.0 <= v < 1.7.1 | No | Yes |
| v >= 1.7.1 | Yes | Yes |

If the rebalancer is running against a cluster where rebalancing replication factor one tables are not supported, it rebalances all the other tables and the cluster as if those singly-replicated tables did not exist.

### Range-aware rebalancing

You can use the Kudu cluster rebalance CLI tool to run range-aware rebalancing. Range-aware rebalancing runs on a per-table basis, which means that the rebalancing can be run on one table at a time. To enable range-aware rebalancing for a particular table, you need to add the following flags when using the CLI tool:

- --enable_range_rebalancing
- --tables=<table_name_for_range_aware_rebalancing>

To perform cluster-wide rebalancing, it is recommended to run the kudu cluster rebalance tool. In addition to the cluster-wide rebalancing, you can run the range-aware rebalancing for larger tables in the cluster that are multilevel-partitioned, and can suffer from the hot-spotting issue.

The default replica placement algorithm is range-aware.

## Run a tablet rebalancing tool on a rack-aware cluster

It is possible to use the kudu cluster rebalance tool to establish the placement policy on a cluster. This might be necessary when the rack awareness feature is first configured or when re-replication violated the placement policy.

### About this task

The rebalancing tool breaks its work into three phases:

### Procedure

1. The rack-aware rebalancer tries to establish the placement policy. Use the ##disable_policy_fixer flag to skip this phase.
2. The rebalancer tries to balance load by location, moving tablet replicas between locations in an attempt to spread tablet replicas among locations evenly. The load of a location is measured as the total number of replicas in the location divided by the number of tablet servers in the location. Use the ##disable_cross_location_rebalancing flag to skip this phase.
3. The rebalancer tries to balance the tablet replica distribution within each location, as if the location were a cluster on its own. Use the ##disable_intra_location_rebalancing flag to skip this phase.

**Note:** Each of the above rebalancing phases can be independently skipped using the corresponding flags.

### What to do next

By using the ##report_only flag, it's also possible to check if all tablets in the cluster conform to the placement policy without attempting any replica movement.

## Example of rebalancing tool with rack awareness functionality

The behavior of each of these flags is explained through the following examples.

Consider three locations and three tablets (a total of 9 replicas) while running the tool with the flags below (placement policy fixer):

```
--disable_cross_location_rebalancing --disable_intra_location_rebalancing
```

Before running the tool:

| Location A | Location B | Location C |
|---|---|---|
| Replica X | Replica Y | Replica Z |
| Replica X | Replica Y | Replica Z |
| Replica X | Replica Y | Replica Z |

After running the tool with the flags:

| Location A | Location B | Location C |
|---|---|---|
| Replica X | Replica X | Replica X |
| Replica Y | Replica Y | Replica Y |
| Replica Z | Replica Z | Replica Z |

The replicas of every tablet are now distributed across all three locations.

Consider the following tablet distribution and run the tool with the flags below (cross-location rebalancing):

```
--disable_policy_fixer --disable_intra_location_rebalancing
```

Before running the tool:

| Location | Number of replicas across all tables in the location |
|---|---|
| A (5 tablet servers) | 15 |

| Location | Number of replicas across all tables in the location |
|---|---|
| B (5 tablet servers) | 18 |
| C (5 tablet servers) | 21 |

After running the tool with the flags:

| Location | Number of replicas across all tables in the location |
|---|---|
| A (5 tablet servers) | 21 |
| B (5 tablet servers) | 21 |
| C (5 tablet servers) | 21 |

The number of replicas in each of the locations is now equal.

Let’s analyze Location A before and after using the tool with the flags (intra-location rebalancing):

```
--disable_policy_fixer --disable_cross_location_rebalancing
```

Before running the tool:

| Tablet server (TS) | Number of replicas across all tables in the server |
|---|---|
| TS_1 | 3 |
| TS_2 | 5 |
| TS_3 | 8 |
| TS_4 | 4 |
| TS_5 | 1 |

After running the tool with the flags:

| Tablet server (TS) | Number of replicas across all tables in the server |
|---|---|
| TS_1 | 4 |
| TS_2 | 5 |
| TS_3 | 4 |
| TS_4 | 4 |
| TS_5 | 4 |

The number of replicas in each tablet server is now balanced.

## Run a tablet rebalancing tool in Cloudera Manager

You access and run the tablet rebalancing tool from Cloudera Manager.

### Procedure

1. Browse to Clusters Kudu .
2. Click Actions and select Run Kudu Rebalancer Tool.

### Results

In Cloudera Manager, the rebalancer runs with the default flags.

## Run a tablet rebalancing tool in command line

Learn how to use Kudu service keytab while running the Kudu tablet rebalancing tool.

### About this task

The following instructions are applicable when Kerberos is enabled in the cluster. If Kerberos is not enabled, you need to only run the following command:

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

### Procedure

1. Run kinit as Kudu user through keytab.

   ```
   $ kinit -kt /var/run/cloudera-scm-agent/process/$(ls -rt /var/run/cloude
   ra-scm-agent/process|grep KUDU|tail -n1)/kudu.keytab kudu/`hostname -f`
   ```

2. Check if kinit ran successfully.

   ```
   $ klist
   ```

3. Run the rebalancing tool as the Kudu admin user, specifying all master addresses.

   ```
   kudu cluster rebalance master-01.example.com,master-02.example.com,maste
   r-03.example.com
   ```

# Managing Kudu tables with range-specific hash schemas

The custom hash partition feature enables you to vary the number of hash buckets per range partition, both at table creation and alteration time.

It is possible to specify a different number of hash buckets per range partition when creating or altering a table. This allows for accommodating new ranges of already existing Kudu tables to changes in workload patterns. For example, for a table range-partitioned by date or timestamp it makes sense to increase the number of hash buckets with an increase in ingestion rate to maximize write throughput.

When no hash schema per range is specified for a table's range either during creation or alteration time, it defaults to the table-wide hash schema. If no table-wide hash schema is specified when the table is created, no hash bucketing for the data is done at all.

Any hash schema specified for a range partition overrides the table-wide hash schema. However, the number of hash dimensions in any range-specific hash schema must be the same as in the table-wide hash schema.

The newly introduced range-specific hash schemas in Kudu are supported by impala-shell.

## Range-specific hash schemas example: Using impala-shell

Review examples of using impala-shell to create and alter Kudu tables with range-specific hash schemas.

The following is a few examples of creating a new table with custom hash schemas for some of its range partitions.

The table-wide hash schema for table t1 has two hash dimensions (hash bucketing by the ‘id’ and the ‘name’ columns independently):

```
CREATE TABLE t1 (id INT, name STRING, PRIMARY KEY(id, name))
PARTITION BY HASH(id) PARTITIONS 3 HASH(name) PARTITIONS 4
RANGE (id)
(
    // hash partition by “id” and “name” as in the table-wide hash schema,
    // but customize the number of hash buckets in each dimension
    PARTITION 15 <= VALUES < 20 HASH(id) PARTITIONS 5 HASH(name) PARTITIONS
 3,
    // this range has the table-wide hash schema
    PARTITION 20 <= VALUES < 25
)
```

The table-wide hash schema for table t2 has one hash dimension (hash bucketing by the ‘name’ column):

```
CREATE TABLE t2 (id INT, name STRING, PRIMARY KEY(id, name))
PARTITION BY HASH(name) PARTITIONS 3
RANGE (id)
(
    // hash partition by “name” but use custom number of hash buckets
    PARTITION  5 <= VALUES < 10 HASH(name) PARTITIONS 5,

    // hash partition by “id”
    PARTITION 10 <= VALUES < 15 HASH(id) PARTITIONS 2,
    // hash partition by “id” and “name”
    PARTITION 15 <= VALUES < 20 HASH(id, name) PARTITIONS 8,

    // this range has the same schema as the range [15, 20) above:
    // hash partition by the primary key (i.e. by (id, name) pair) into 8
 buckets
    PARTITION 20 <= VALUES < 25 PARTITIONS 8,

    // using the table-wide hash schema if no override is specified
    PARTITION 25 <= VALUES < 30
)
```

The following are examples of adding new range partitions with custom hash schemas to a table that already exists:

```
// t2: add new range partition by “id”, hashed by “id” and “name” together
ALTER TABLE t2 ADD RANGE PARTITION -5 <= VALUES < 0 HASH PARTITIONS 10;

// t1: add new range partition by “id”, additionally hashed by “id” and “nam
e” separately
// (defaults to the table-wide hash schema since no range-specific hash sc
hema specified)
ALTER TABLE t1 ADD RANGE PARTITION 25 <= VALUES < 30

// t2: add new range partition by “id”, additionally hashed by “name”
ALTER TABLE t2 ADD RANGE PARTITION 30 <= VALUES < 40 HASH(name) PARTITIONS 5
;

// t1: add new range partition by “id”, additionally hashed by “id” and “
name” separately
ALTER TABLE t1 ADD RANGE PARTITION 40 <= VALUES < 50 HASH(id) PARTITIONS 10,
 HASH(name) PARTITIONS 5;
```

## Range-specific hash schemas example: Using Kudu C++ client API

Review an example of using Kudu C++ client API to create a table with a range partition having range-specific hash schema and then alter the table, adding one more range with custom hash schema.

**Note:** After adding or dropping ranges with custom hash schemas you have to refresh the KuduTable handle.

```
 // Define the column schema for the table.
  KuduSchema schema;
  {
    KuduSchemaBuilder b;
    b.AddColumn("key")->Type(KuduColumnSchema::INT32)->NotNull()->PrimaryK
ey();
    b.AddColumn("int_col")->Type(KuduColumnSchema::INT32)->NotNull();
    b.AddColumn("string_col")->Type(KuduColumnSchema::STRING);
    RETURN_NOT_OK(b.Build(&schema));
  }
  unique_ptr<KuduTableCreator> tc(client->NewTableCreator());
  // The table is range-partitioned by the ‘key’ column and has table-wide
 hash schema:
  // one-dimensional hash bucketing into two buckets by the ‘key’ column.
  table_creator->table_name("test")
      .schema(&schema)
      .add_hash_partitions({ "key" }, 2)
      .set_range_partition_columns({ "key" });
  // Add [0, 100) range partition with custom hash schema:
  // 5 buckets with hash based on the "key" column with hash seed 8.
  {
    unique_ptr<KuduPartialRow> lower(schema.NewRow());
    RETURN_NOT_OK(lower->SetInt32("key", 0));

    unique_ptr<KuduPartialRow> upper(schema.NewRow());
    RETURN_NOT_OK(upper->SetInt32("key", 100));

    unique_ptr<KuduRangePartition> p(
        new KuduRangePartition(lower.release(), upper.release()));
    RETURN_NOT_OK(p->add_hash_partitions({ "key" }, 5, 8));
    tc->add_custom_range_partition(p.release());
  }

  // Create the ‘test’ table with the specified column schema
  // and one range partition.
  RETURN_NOT_OK(tc->Create());

  // Open the table and work with it.
  shared_ptr<KuduTable> table;
  RETURN_NOT_OK(client->OpenTable("test", &table));
  . . .

  // Alter the table, adding a new range [100, 200) with range-specific hash
  // schema: 3 buckets with hash based on the "key" column with hash seed 1.
  {
    unique_ptr<KuduPartialRow> lower(schema.NewRow());
    RETURN_NOT_OK(lower->SetInt32("key", 222));

    unique_ptr<KuduPartialRow> upper(schema.NewRow());
    RETURN_NOT_OK(upper->SetInt32("key", 333));

    unique_ptr<KuduRangePartition> p(
```

```
        new KuduRangePartition(lower.release(), upper.release()));
  RETURN_NOT_OK(p->add_hash_partitions({ "key" }, 3, 1));

  unique_ptr<KuduTableAlterer> ta(client->NewTableAlterer(kTableName));
  ta->AddRangePartition(p.release());
  RETURN_NOT_OK(ta->Alter());
}

// NOTE: after adding or dropping a range with custom hash schemas
//       it's necessary to refresh the KuduTable handle
RETURN_NOT_OK(client->OpenTable("test", &table));

// Continue working with the table.
. . .
```

## Range-specific hash schemas example: Using Kudu Java client API

Review an example of using Kudu Java client API to create a table with a range partition having range-specific hash schema and then alter the table, adding one more range with custom hash schema.

**Note:** After adding or dropping ranges with custom hash schemas you have to refresh the KuduTable handle.

```
// Define the column schema.
ArrayList<ColumnSchema> columns = new ArrayList<>(2);
columns.add(new ColumnSchema.ColumnSchemaBuilder(
    "key", Type.INT32).key(true).build());
columns.add(new ColumnSchema.ColumnSchemaBuilder(
    "col", Type.INT64).build());
final Schema schema = new Schema(columns);
CreateTableOptions builder = new CreateTableOptions();

// Do range partitioning on the "key" column.
builder.setRangePartitionColumns(ImmutableList.of("key"));
// Define the table-wide schema: two buckets on the "key" column.
builder.addHashPartitions(ImmutableList.of("key"), 2, 0);

// Add [0, 100) range partition with custom hash schema.
{
  PartialRow lower = schema.newPartialRow();
  lower.addInt("key", 0);
  PartialRow upper = schema.newPartialRow();
  upper.addInt("key", 100);

  // Custom hash schema for the range: five buckets on the "key" column.
  RangePartitionWithCustomHashSchema rangePartition =
      new RangePartitionWithCustomHashSchema(
          lower,
          upper,
          RangePartitionBound.INCLUSIVE_BOUND,
          RangePartitionBound.EXCLUSIVE_BOUND);
  rangePartition.addHashPartitions(ImmutableList.of("key"), 5, 0);
  builder.addRangePartition(rangePartition);
}

// Create the table.
KuduTable table = client.createTable("mytable", schema, builder);

// Work with the table.
```

```
  . . .

  // Alter the table: add [100, 200) range partition with custom hash sche
ma.
  {
    PartialRow lower = schema.newPartialRow();
    lower.addInt("key", 100);
    PartialRow upper = schema.newPartialRow();
    upper.addInt("key", 200);
    RangePartitionWithCustomHashSchema range =
        new RangePartitionWithCustomHashSchema(
            lower,
            upper,
            RangePartitionBound.INCLUSIVE_BOUND,
            RangePartitionBound.EXCLUSIVE_BOUND);
    range.addHashPartitions(ImmutableList.of("key"), 3, 0);
    client.alterTable("mytable",
        new AlterTableOptions().addRangePartition(range));
  }

  // NOTE: after adding or dropping a range with custom hash schemas
  //       it's necessary to refresh the KuduTable handle
  table = client.openTable("mytable");

  // Continue working with the table.
  . . .
```

# Manage one-dimensional arrays in Kudu

Kudu supports one-dimensional arrays of scalar types, such as integers, strings, booleans, and binaries, directly within columns. This feature enables more expressive schema designs and improves query performance by removing the need to denormalize data or manage child tables with complex joins.

## Key Benefits

- Previously, array data, such as device telemetry, required splitting into numerous columns by index to bypass Kudu's default 300-column limit. You can now store arrays natively in a single column, which simplifies the schema and avoids complex denormalization.
- You do not need to reassemble split data during read operations because applications can use native array types directly.

## System limits

- Hard limit: Each array cell is restricted to a maximum of 65,535 elements due to storage encoding constraints.
- Configurable limits: Administrators can restrict the maximum number of array elements in a column cell by using the --array_cell_max_elem_num flag. This flag is set to 1024 by default. Setting this limit mitigates performance issues and enforces storage policies. If a client application submits an operation to a Kudu tablet server that exceeds this limit, the server returns an error to the client.

**Important:** Creating array columns is a behavior change that affects backward compatibility. You cannot roll back to earlier Kudu versions after creating a table with an array type column because earlier versions cannot read the columnar array block format. Tables that do not contain arrays are not affected by this limitation.

# Kudu client API for data operations

The Kudu client APIs allow you to insert and manage arrays in tables using various programming languages.

The Kudu client supports the following languages:

- C++
- Java
- Scala and Spark bindings
- Python

## Working with array column data in C++

The following code demonstrates how to manage array column data using the Kudu C++ client API.

### Setting array column data in C++

The Kudu C++ client API allows you to set, retrieve, and manage array column values.

**Setting array column data in C++**

The Kudu C++ client API allows you to set, retrieve, and manage array column values.

- Setting array column values: The following example shows how to set an INT64 array column named arr_int64 to an array of three elements. This includes two non-null elements and one null element in the middle, for example, [6, NULL, 7]:

```
// KuduPartialRow* row ...;
KUDU_RETURN_NOT_OK(row->SetArrayInt64("arr_int64", { 6, 0, 7
 }, { true, false, true }));
```

- Setting a column to NULL: To set the entire array column to a null value:

```
KUDU_RETURN_NOT_OK(row->SetNull("arr_int64"));
```

- Setting an empty array: To set the column to an empty array, for example []:

```
KUDU_RETURN_NOT_OK(row->SetArrayInt64("arr_int64", {}, {}));
```

- Setting an array with a single null element: To set an array containing one null element, for example [NULL]:

```
KUDU_RETURN_NOT_OK(row->SetArrayInt64("arr_int64", { 0 }, {
false }));
```

- Setting an array with all non-null elements: There is a convention to accept an empty validity vector when all array elements are non-null. The same convention applies when retrieving data from array type columns.

```
KUDU_RETURN_NOT_OK(row->SetArrayInt32("arr_int32", { 1, 2, 3
 }, {}))
```

**Retrieving array column data in C++**

The following example shows how to access the value of an INT64 array column in a row scan projection. This example uses column index instead of column name, assuming an INT64 array column is at index 3 of the row scan projection's schema:

```
// KuduScanBatch::const_iterator it = ...;
KuduScanBatch::RowPtr row(*it);
std::vector<int64_t> arr;
```

```
std::vector<bool> arr_notnull;
KUDU_RETURN_NOT_OK(row.GetArrayInt64(3, &arr, &arr_notnull));
```

**Accessing raw array cell data:**

In the Kudu C++ client API, it's also possible to access raw data in an array cell. The following example shows how to access raw array cell data using the KuduArrayCellView facade.

```
const void* cell_ptr = row.cell(3);
KuduArrayCellView view(cell_ptr);
KUDU_RETURN_NOT_OK(view.Init());
KUDU_CHECK(view.elem_num() == arr.size());
const uint8_t* arr_raw_notnull_bitmap = view.not_null_bitmap();
KUDU_CHECK(arr_raw_notnull_bitmap);
const int64_t* arr_raw = reinterpret_cast<const int64_t*>(view
.data(KuduColumnSchema::INT64));
KUDU_CHECK(arr_raw);
for (size_t i = 0; i < arr_notnull.size(); ++i) {
  if (arr_notnull[i]) {
    KUDU_CHECK(arr[i] == *(arr_raw + i));  // that's the same da
ta for non-null array elements as when accessing it via RowPtr::
GetArrayInt64()
  }
  bool arr_raw_elem_validity = *(arr_raw_notnull_bitmap + (i >>
3)) & (1 << (i & 7)); // the validity bit directly corresponds t
o the boolean value at the corresponding index of the 'arr_notnu
ll' validity array
  KUDU_CHECK(arr_notnull[i] == arr_raw_elem_validity);
}
```

## Working with array column data in Java

The Kudu Java client allows you to set and manage array column values using primitive arrays (long[]) and validity arrays (boolean[]).

### Setting array column data in Java

**Setting array column values**

To set an INT64 array column named arr_int64 to [1L, NULL, 3L], provide the values List<Long> containing a null entry:

```
import org.apache.kudu.client.PartialRow;
// PartialRow row ...;
long[] vals = {1L, 2L, 3L};
boolean[] validity = {true, false, true};
row.addArrayInt64("arr_int64", vals);
```

- Setting a column to NULL: To set the entire column (the array itself) to a null value:

  ```
  row.setNull("arr_int64");
  ```

- Setting an empty array: To set the column to an empty array (e.g., []):

  ```
  long[] values = {};
  row.addArrayInt64("arr_int64", values);
  ```

- Setting an array with a single null element: To set an array containing exactly one null element (e.g., [NULL]):

  ```
  long[] values = { 0L };
  boolean[] validity = { false };
  ```

```
row.addArrayInt64("arr_int64", values, validity);
```

- Setting an array with all non-null elements: If the array contains no null values, you can use a convenience overload that omits the validity vector entirely. Kudu will automatically treat all elements in the provided array as non-null.

```
long[] values = { 10L, 20L, 30L };
row.addArrayInt64("arr_int64", values);
```

**Retrieving array column data in Java**

The following example shows how to access an INT64 array column using either the column name or the column index:

```
while (scanner.hasMoreRows()) {
  RowResultIterator it = scanner.nextRows();
  while (it.hasNext()) {
    rows.add(it.next());
  }
}
RowResult r1 = rows.get(0);
Long[] int64Vals = (Long[]) r1.getArrayData("arr_int64");
```

## Working with array column data in Spark bindings

Learn how Kudu Spark bindings treat Kudu arrays as standard Spark ArrayType and provides examples for creating and accessing DataFrame with array columns.

Kudu Spark bindings treat Kudu arrays as standard Spark ArrayType (mapped to Seq in Scala).

### Creating DataFrame with array columns

This example demonstrates how to create a DataFrame in Apache Spark with array columns, including handling null elements within arrays.

```
import org.apache.spark.sql.functions._
import spark.implicits._
// Sample data with arrays (including null elements)
val df = Seq(
  (1, Seq(1, null, 3), Seq("a", null, "c")),
  (2, Seq(4, 5, 6), Seq("x", "y", "z"))
).toDF("key", "int_arr", "str_arr")
```

### Accessing array column data

You can use standard SQL index notation to access elements.

```
val rows = df.select("key", "int_arr", "str_arr").collect()
val first = rows(0)
val ints = first.getAs[Seq[Integer]]("int_arr")
val strs = first.getAs[Seq[String]]("str_arr")
println(ints) // Seq(1, null, 3)
println(strs) // Seq("a", null, "c")
```

### Using Spark SQL to access array column data

This section demonstrates how to use Spark SQL to access and manipulate array column data in a DataFrame.

```
df.createOrReplaceTempView("t")
spark.sql("""
  SELECT key, int_arr[0] AS first_elem
  FROM t
  ORDER BY key
""").show()

+---+----------+
|key|first_elem|
+---+----------+
|  1|         1|
|  2|         4|
+---+----------+
```

## Working with array column data in Python

Learn how to work with array column data using the Kudu Python client.

The Kudu Python client maps Kudu arrays directly to Python lists, where None represents a null element.

### Setting array column values

When defining a schema or inserting rows, use kudu.array_type and standard Python lists.

**Adding an array column to a schema:**

```
builder = kudu.schema_builder()
builder.add_column('arr_int64').nested_type(kudu.array_type(kud
u.int64))
```

**nserting a row with nulls [6, NULL, 7]:**

```
op = table.new_insert()
op['arr_int64'] = [6, None, 7]
session.apply(op)
```

**Setting an empty array [] or a NULL column:**

- Empty Array: op['arr_int64'] = []
- NULL Column: op['arr_int64'] = None

### Retrieving array column data

When selecting data, array columns are returned as Python lists.

```
scanner = client.new_scanner('my_table')
scanner.open()
while scanner.has_more_rows():
    batch = scanner.next_batch()
    for row in batch:
        print(row['arr_int64']) # Output: [6, None, 7]
```

## Impala integration with Kudu arrays

Learn about using Impala to query Kudu array columns and the limitations of this integration.

You can use Impala to query Kudu array columns by using standard array syntax compatible with other formats, such as Parquet.

### Limitations of Impala Integration

The following limitations apply to the integration of Impala with Kudu:

- **Read-only access:** Impala currently cannot write array data to Kudu. To write array data, you must use the Kudu client API.
- **One-dimensional arrays only:** Although the Imapa SQL engine supports multi-dimensional arrays, the Kudu integration supports only one-dimensional arrays. This is because Kudu storage currently only supports one-dimensional arrays.

# Replicating Kudu tables using Apache Flink

Learn about using the Apache Flink-based replication job to continuously replicate data from a source Kudu cluster to a sink Kudu table.

The Kudu replication job is an Apache Flink streaming application that continuously copies data from a source Kudu table to a sink Kudu table. It uses the Kudu diff scan capability to detect and replicate only the rows that changed between two points in time. This is more efficient than scanning the full table on every cycle.

### Replication mechanism

The replication mechanism follows these steps:

1. On the first startup, the job performs a full snapshot scan of the source table and records the snapshot timestamp t0.
2. On each subsequent discovery cycle, it performs a diff scan over the interval [t_previous, t_now]. This returns all inserted, updated, and deleted rows in that window.
3. Deleted rows are identified by using the Kudu IS_DELETED virtual column and are replicated as delete operations on the sink.
4. All writes to the sink use upsert-ignore semantics, which makes replication idempotent and compatible with at-least-once Flink checkpointing.

Flink checkpoints occur regularly to enable fault-tolerant recovery. If a restart occurs from a checkpoint, any splits that were in-flight at the time of the checkpoint are reprocessed. Because sink writes are idempotent, reprocessing is safe.

### Known limitations

- Schema changes: These are not applied automatically. You must follow the manual schema change procedure strictly.
- Unsupported types: The ARRAY column type is not currently supported. You cannot replicate tables containing array columns.
- Table renames: Renaming a replicated table requires a specific coordinated procedure.

## Prerequisites to start replication job

Ensure the necessary requirements and configurations are in place before starting the replication job.

### About this task

You can start the replication job by submitting the shaded JAR to a Flink cluster using the standard `flink run` command. All configuration must be passed as --key value command-line arguments.

### Before you begin

Before starting the replication job, ensure the following requirements exist:

- A running Apache Flink cluster in session or per-job mode.
- A source Kudu cluster that contains the table you intend to replicate.
- A sink Kudu cluster where the destination table is already created, or where job.createTable=true is set to enable the job to create the table automatically.
- A shared filesystem, such as HDFS or S3, that is accessible by all Flink TaskManagers for storing Flink checkpoints.
- The replication job JAR file (kudu-replication-<version>.jar). This is a shaded JAR file that includes all Kudu dependencies. The Flink cluster provides the required Flink APIs at runtime.

### Source cluster configuration

### Procedure

1. Configure the source cluster to support diff scans.

   To support diff scans, the Kudu --tablet_history_max_age_sec property (default 7 days) on tablet servers must cover the maximum expected downtime for the replication job. If the job remains stopped for longer than this period, diff scans can fail because Kudu garbage collects the required history.

   **Important:** Ensure this property is set to a duration that matches or exceeds your recovery time objectives.

2. Verify the master server configuration.

   The master server also uses the --tablet_history_max_age_sec property with a default of 5 minutes, but this setting does not affect diff scanning operations.

## Security for Kudu replication

Learn about configuring Kerberos authentication, Ranger authorization, and classloader isolation for the Kudu replication job.

### Kerberos authentication

The Kudu Java client negotiates Kerberos authentication automatically by using the ambient Kerberos ticket cache through GSSAPI or JAAS. The replication job configuration contains no explicit keytab or principal properties.

When you run the job on YARN, you must follow these requirements:

1. Ensure a valid Kerberos ticket exists for the submitting user on the node where you run the `flink run` command.
2. YARN propagates Hadoop delegation tokens to the JobManager and TaskManager containers automatically.
3. For long-running jobs where the environment cannot guarantee ticket renewal, you can use the built-in filnk keytab support. To provide credentials explicitly, pass the following -D flags at submission time:

```
flink run-application -t yarn-application \
  -Dsecurity.kerberos.login.keytab=/path/to/user.keytab \
  -Dsecurity.kerberos.login.principal=user@REALM \
  -Dclassloader.parent-first-patterns.additional=org.apache.kudu \
  -c org.apache.kudu.replication.ReplicationJob \
```

```
kudu-replication-<version>.jar
```

### Authorization using Ranger

If the Kudu cluster uses Ranger integration, the user submitting the replication job must have the following privileges:

| Resource | Required privilege |
|---|---|
| Source table | select, metadata |
| Sink database | create (only if job.createTable=true) |
| Sink table | all |

### Sink cluster protection

Cloudera recommends restricting write access on the sink cluster so that only the replication service user can write to replicated tables. This restriction prevents accidental data manipulation on the disaster recovery (DR) target and avoids data divergence between the source and sink.

The following table shows the recommended Ranger policy setup for the sink cluster:

| Principal | Resource | Privilege | Rationale |
|---|---|---|---|
| Replication service user | Sink tables | all | Enables the replication job to perform upsert and delete operations. |
| All other users | Sink tables | select, metadata | Enables read-only queries without the risk of accidental writes. |
| All other users | Sink tables | None | Explicitly denies or omits insert, update, and delete privileges. |

**Important:** You can use Ranger deny policies to block write operations for all users except the replication service user. This approach is the most secure for a DR cluster.

### Classloader isolation

To avoid classloader conflicts between the Kudu client and the Flink child-first classloader, you must pass the following argument during job submission:

```
-Dclassloader.parent-first-patterns.additional=org.apache.kudu
```

This property ensures that the same parent classloader loads both the Kudu classes and the Hadoop security classes, which prevents ClassCastException errors during Kerberos negotiation.

## Starting the replication job

Learn how to start a Kudu replication job and how to resume a job from a specific checkpoint.

### Starting a new replication job

You can start the replication job by submitting the shaded JAR file to a Flink cluster by using the `flink run-application` command. You must pass all configuration parameters as --key value flags.

To start a basic replication job on YARN, run the following command:

```
flink run-application -t yarn-application \
  -Dclassloader.parent-first-patterns.additional=org.apache.kudu \
  -c org.apache.kudu.replication.ReplicationJob \
  kudu-replication-<version>.jar \
```

```
  --job.sourceMasterAddresses source-master1:7051,source-master2:7051 \
  --job.sinkMasterAddresses sink-master1:7051,sink-master2:7051 \
  --job.tableName my_table \
  --job.checkpointsDirectory hdfs:///kudu-replication/checkpoints/my_table
```

### Resuming from a checkpoint

If a job fails or you perform a planned restart, you can resume from a specific Flink checkpoint or savepoint. Flink retains checkpoints on cancellation by default by using the RETAIN_ON_CANCELLATION setting.

To resume the job, run the following command and include the -s flag with the path to the checkpoint directory:

```
flink run-application -t yarn-application \
  -s hdfs:///kudu-replication/checkpoints/my_table/<checkpoint-id> \
  -Dclassloader.parent-first-patterns.additional=org.apache.kudu \
  -c org.apache.kudu.replication.ReplicationJob \
  kudu-replication-<version>.jar \
  --job.sourceMasterAddresses source-master1:7051,source-master2:7051 \
  --job.sinkMasterAddresses sink-master1:7051,sink-master2:7051 \
  --job.tableName my_table \
  --job.checkpointsDirectory hdfs:///kudu-replication/checkpoints/my_table
```

## Stopping the replication job

Learn how to safely stop the Kudu replication job by using a savepoint to ensure a consistent state for future restarts.

### Stopping the job by using a savepoint

You must always stop the replication job by using a savepoint. A savepoint is a consistent snapshot of the entire job state, including the enumerator position, in-flight splits, and the lastEndTimestamp property, which the system writes to durable storage.

Stopping the job with a savepoint allows you to restart the job later from exactly where it left off. This process prevents the need to re-run the initial full snapshot scan, saving time and resources.

## Kudu replication Pre-stop checklist

Use the following checklist to ensure the Kudu replication pipeline is fully drained and no data is lost before you stop the job.

### About this task

Before you stop the replication job for reasons such as a schema change, planned failover, or maintenance, you must follow this sequence to ensure the pipeline is fully drained.

### Procedure

1. Stop all writes to the source table.

   This is an application-level step. You must quiesce or redirect the write path before you proceed.
2. Verify no writes are in-flight on the source cluster.

   Run the write activity PromQL query described in the topic. Proceed only when the query returns 0.
3. Wait for replication lag to normalize.

   Run the replication lag query. The lag must reach approximately the value of the job.discoveryIntervalSeconds property divided by 60 minutes. This confirms that the final diff scan captured all pending changes.

4. Verify the sink cluster has fully drained.

   Ensure the pendingCount, unassignedCount, and pendingRemovalCount metrics are all 0 between discovery cycles.

5. Verify no write activity exists on the sink cluster.

   Run the write activity PromQL query for the sink table to confirm that the replication job has flushed all buffered operations. Proceed only when the query returns 0.

6. Stop the job by using a savepoint.

   Run the following command:

   ```
   flink stop -p hdfs:///kudu-replication/savepoints/my_table <job-id>
   ```

   The command displays the savepoint path upon completion:

   ```
   Savepoint completed. Path: hdfs:///kudu-replication/savepoints/my_table/
   savepoint-<id>
   ```

### Results

**Important:** The `flink stop` command waits for checkpoint barriers to propagate through the pipeline. If the job is processing a large discovery cycle, the stop operation can experience a delay.

## Restarting replication job and finding job ID

Learn how to restart the Kudu replication job from a saved state and how to identify the job ID on YARN or a session cluster.

### Restarting from a savepoint

To restart the replication job from a savepoint, you must pass the savepoint path by using the -s flag during job submission.

Run the following command to restart the job:

```
flink run-application -t yarn-application \
  -s hdfs:///kudu-replication/savepoints/my_table/savepoint-<id> \
  -Dclassloader.parent-first-patterns.additional=org.apache.kudu \
  -c org.apache.kudu.replication.ReplicationJob \
  kudu-replication-<version>.jar \
  --job.sourceMasterAddresses source-master1:7051,source-master2:7051 \
  --job.sinkMasterAddresses   sink-master1:7051,sink-master2:7051 \
  --job.tableName              my_table \
  --job.checkpointsDirectory  hdfs:///kudu-replication/checkpoints/my_table
```

### Finding the job ID

You must identify the job ID to perform management tasks such as stopping the job or creating a savepoint.

- On YARN: To find the job ID of a running replication job on YARN, run the following command:

  ```
  flink list -t yarn-application -Dyarn.application.id=<yarn-application-id>
  ```

- On a session cluster: To list all running jobs across a session cluster, run the following command:

# Kudu replication configuration reference

Use the following parameters to configure the job, reader, and writer settings for Kudu replication.

## Job Parameters

The following table describes the parameters that control the overall replication behavior:

| Parameter | Required | Default | Description |
|---|---|---|---|
| job.sourceMasterAddresses | Yes | N/A | The comma-separated list of source Kudu master addresses, such as host1:7051,host2:7051. |
| job.sinkMasterAddresses | Yes | N/A | The comma-separated list of sink Kudu master addresses. |
| job.tableName | Yes | N/A | The name of the source Kudu table to replicate. |
| job.checkpointsDirectory | Yes | N/A | The filesystem path where Flink stores checkpoint data, such as an HDFS path. |
| job.discoveryIntervalSeconds | No | 600 | The frequency in seconds that the job polls for new changes by using a diff scan. |
| job.checkpointingIntervalMillis | No | 60000 | The interval in milliseconds at which Flink takes checkpoints. This value must be strictly less than the result of job.discoveryIntervalSeconds * 1000. |
| job.createTable | No | false | If set to true, the job automatically creates the sink table if it does not exist. |
| job.tableSuffix | No | "" | The suffix appended to the sink table name. This is useful for testing replication to a table with a different name on the same cluster. |
| job.restoreOwner | No | false | If set to true and job.createTable=true, the job copies the table owner from the source to the sink table. |

**Important:** The job.checkpointingIntervalMillis property must be strictly less than the product of job.discoveryIntervalSeconds and 1000. This ensures that the replication timestamp advances correctly.

## Reader Parameters

The following table describes the parameters that control how the job reads data from the source Kudu cluster:

| Parameter | Required | Default | Description |
|---|---|---|---|
| reader.batchSizeBytes | No | 20971520 | The maximum number of bytes fetched in a single scan batch (default is 20 MiB). |
| reader.splitSizeBytes | No | Kudu default | The target size in bytes for each scan split when parallelizing input. |
| reader.scanRequestTimeout | No | 30000 ms | The timeout in milliseconds for individual scan RPCs. |

| Parameter | Required | Default | Description |
|---|---|---|---|
| reader.prefetching | No | false | Whether to enable prefetching of data blocks from the scanner. |
| reader.keepAlivePeriodMs | No | 15000 ms | The period in milliseconds after which an idle scanner sends a keep-alive message to the server. |
| reader.replicaSelection | No | CLOSEST_REPLICA | The replica selection strategy.<br>Valid values :<br>CLOSEST_REPLICA or LEADER_ONLY. |

### Writer Parameters

The following table describes the parameters that control how the job writes data to the sink Kudu cluster:

| Parameter | Required | Default | Description |
|---|---|---|---|
| writer.flushMode | No | AUTO_FLUSH_BACKGROUND | The Kudu session flush mode.<br>Valid values :<br>AUTO_FLUSH_BACKGROUND, AUTO_FLUSH_SYNC, or MANUAL_FLUSH. |
| writer.operationTimeout | No | 30000 ms | The timeout in milliseconds for individual write operations. |
| writer.maxBufferSize | No | 1000 | The maximum number of operations buffered in the Kudu write session. |
| writer.flushInterval | No | 1000 ms | The interval in milliseconds at which the job automatically flushes buffered operations. |

## Supported Kudu data types for replication

Refer to the table below to identify the Kudu column types that are supported for replication using the Flink-based job.

**Important:** The ARRAY column type is not currently supported. You cannot replicate tables that contain ARRAY columns.

### Table 1: Supported Kudu data types

| Kudu type | Notes |
|---|---|
| STRING | UTF-8 encoded text. |
| VARCHAR | The Flink connector maps this to STRING. The length constraint is not enforced. |
| BOOL | Supported. |
| INT8, INT16,INT32, INT64 | Supported. |
| FLOAT, DOUBLE | Supported. |
| DECIMAL(precision, scale) | Precision and scale are preserved during replication. |
| UNIXTIME_MICROS | Timestamps with microsecond precision. |
| BINARY | Supported. |

# Handling schema changes for Kudu replication

Learn how to manually apply schema changes to replicated Kudu tables while maintaining data consistency between source and sink clusters.

**Warning:** The schema change procedure depends on monitoring to verify specific preconditions. Without a working Prometheus configuration, no reliable method exists to determine when you can safely apply Data Definition Language (DDL) changes. Applying a schema change while writes are still in flight can produce rows with mismatched schemas on the sink cluster.

## Schema change requirements

The replication job does not automatically detect or apply schema changes. If the schema of the source table changes, such as when a column is added, you must perform the change manually on both the source and sink tables. After applying the changes, you must restart the replication job from a savepoint.

# Adding column and performing schema changes

Learn how to add a column to replicated Kudu tables or perform other schema modifications, such as dropping or renaming columns, by using a coordinated manual process.

## About this task

This topic provides instructions for adding a column to replicated Kudu tables and performing schema modifications.

## Procedure

1. Follow the Kudu replication Pre-stop checklist Completing the replication pre-stop checklist to drain the pipeline and stop the replication job by using a savepoint.
2. Add the new column to the source Kudu table by running the following command:

```
kudu table add_column <source-master-addresses> my_table new_column INT32
 \
  --default 0
```

3. Add the same column with the same definition to the sink Kudu table by running the following command:

```
kudu table add_column <sink-master-addresses> my_table new_column INT32 \
  --default 0
```

4. Restart the replication job from the savepoint. For more information, see Restarting the replication job and finding the job ID.

## What to do next

**Important:** The new column must have a default value so that the system can read rows inserted before the column was added without errors during the next diff scan.

**Tip:** For column type changes, you must ensure the new type is compatible with any data already written to the sink cluster.

# Renaming replicated table

Learn how to rename a Kudu table involved in a replication job by performing coordinated changes on both the source and sink clusters.

### About this task

Renaming a replicated table requires coordinated changes on both clusters and a controlled job restart. You must stop the job cleanly by ensuring no writes are in flight before you rename any table.

### Procedure

1. Follow the Kudu replication Pre-stop checklist to drain the pipeline.
2. Stop the replication job by using a savepoint.

   You must confirm that the pendingCount, unassignedCount, and pendingRemovalCount metrics are all zero before you stop the job. This ensures the savepoint contains no in-flight scan splits.
3. Rename the source Kudu table by running the following command:

   ```
   kudu table rename_table <source-master-addresses> old_name new_name
   ```
4. Rename the sink Kudu table.

   If the job.tableSuffix property is configured, the sink table name follows the <tableName><suffix> format. You must rename it to new_name<suffix> to match the new source name.

   Without a suffix:

   ```
   kudu table rename_table <sink-master-addresses> old_name new_name
   ```

   With the job.tableSuffix=_dr property (for example):

   ```
   kudu table rename_table <sink-master-addresses> old_name_dr new_name_dr
   ```
5. Restart the replication job from the savepoint.

   For more information, see Restarting the replication job and finding the job ID. You must pass the new table name by using the --job.tableName new_name argument. The job.tableSuffix value does not change.

### Results

**Note:** When you stop the job cleanly, the only durable state in the savepoint is the lastEndTimestamp property. This timestamp is independent of the table name, which allows the job to resume incremental diff scanning correctly after the rename.

**Warning:** You must rename both the source and sink tables. Renaming only one table causes the replication job to fail or replicate data to the incorrect table.

## Resource configuration for Kudu replication

Learn how to size resources for Kudu replication across three independent dimensions: the Kudu source cluster, the Kudu Java client, and the Flink cluster.

### Sizing Kudu ource cluster

A longer --tablet_history_max_age_sec window means Kudu tablets accumulate more UNDO delta data before the system can perform garbage collection. You must monitor the following tablet server flag to ensure background maintenance stays current:

| Flag | Default | Description |
|---|---|---|
| --maintenance_manager_num_threads | 1 | Specifies the number of threads for all background maintenance operations, such as flush, compaction, and UNDO garbage collection. With a large history window, the default thread can fall behind. The recommended ratio is 1 thread per 3 data directories. |

## Tuning replication job Kudu java client

The reader.* and writer.* parameters are the primary interface for tuning the Kudu Java client embedded in the replication job. The following parameters have the most significant impact on throughput and memory use:

| Parameter | Default | Technical impact |
|---|---|---|
| reader.batchSizeBytes | 20971520 (20 MiB) | Controls the amount of data each scan RPC fetches. You can increase this value for large tables on fast networks or decrease it to reduce per-reader heap usage. |
| reader.splitSizeBytes | Kudu default | Smaller values produce finer-grained splits, which increases parallelism but requires more scan RPCs. |
| writer.maxBufferSize | 1000 | The number of operations buffered client-side before a flush occurs. Increase this value for higher write throughput; note that each buffered operation consumes heap memory. |
| writer.flushInterval | 1000 ms | Specifies how often buffered writes are flushed. Lower values reduce latency, while higher values improve batching efficiency. |

## Sizing flink cluster

The replication job is a two-operator pipeline (source to sink) with no intermediate transformations. Resource requirements are proportional to the configured parallelism.

**Deployment modes**

The job runs on YARN in one of two modes:

- Application mode: This is the recommended mode for Flink 1.15 and later.
- Per-job mode: This mode is deprecated since Flink 1.15 but remains supported.

In both modes, YARN spawns dedicated containers for the JobManager and TaskManagers. You must ensure that sufficient YARN resources, such as memory and vcores, are available if the cluster is shared with other workloads like Spark or Impala.

**Parallelism and memory**

You set parallelism at submission time by using the -p flag. Each parallel instance runs one Kudu reader and one Kudu writer.

- Parallelism limits: The effective upper bound on useful parallelism is the number of scan tokens Kudu produces for the table. Setting -p higher than the tablet count does not improve performance because extra reader instances remain idle. To raise this ceiling, you can reduce the reader.splitSizeBytes property to divide large tablets into multiple tokens.
- TaskManager memory: Sizing depends on row width, parallelism, and slot count. The main memory drivers are the scan batch buffer and the write buffer. If YARN kills TaskManagers due to out-of-memory (OOM) errors, you must increase the taskmanager.memory.process.size property or reduce the reader.batchSizeBytes property.
- JobManager memory: The footprint for the enumerator is small regardless of table size. The Flink default is sufficient for most scenarios.

Example submission command:

To apply these settings, pass them as -D flags during job submission:

```
flink run-application -t yarn-application \
  -p 4 \
  -Dtaskmanager.numberOfTaskSlots=2 \
  -Dclassloader.parent-first-patterns.additional=org.apache.kudu
\
  -c org.apache.kudu.replication.ReplicationJob \
  kudu-replication-<version>.jar \
  --job.sourceMasterAddresses source-master1:7051 \
  --job.sinkMasterAddresses   sink-master1:7051 \
  --job.tableName              my_table \
  --job.checkpointsDirectory  hdfs:///kudu-replication/checkpoint
s/my_table
```

# Failover and monitoring for Kudu replication

Learn how to perform a disaster recovery cutover, monitor replication health by using Flink metrics, and configure Prometheus and Grafana for visualization.

When you use a replicated sink cluster as a disaster recovery (DR) target, you must follow these procedures to promote the sink cluster to active status.

## Planned failover (Source cluster available)

Use this procedure if the source cluster is reachable and you must cut over with minimal data loss.

1. Follow the Pre-stop checklist to drain the pipeline.
2. Stop the replication job by using a savepoint.
3. Enable write access on the sink cluster by updating the Ranger policy.
4. Redirect application traffic to the sink cluster by updating DNS records or connection strings.

**Tip:** You do not need to restart the replication job unless you intend to perform reverse replication.

## Unplanned failover (Source cluster unavailable)

Use this procedure if the source cluster fails unexpectedly.

1. Assess the replication lag at the time of failure by checking the lastEndTimestamp value in Grafana or Prometheus. The data loss window equals the replication lag at the time of failure.
2. Stop the replication job. If the job is unreachable, kill the YARN application by using the ResourceManager UI or the yarn application -kill command.
3. Enable write access on the sink cluster by updating the Ranger policy.
4. Redirect application traffic to the sink cluster.

## Kudu replication metrics reference

The replication job exposes custom metrics through the Flink metrics system. In a steady state, all split metrics must be zero. A continuously advancing lastEndTimestamp indicates healthy replication.

| Metric Name | Type | Description |
|---|---|---|
| lastEndTimestamp | Gauge (Long) | The end timestamp of the most recently completed diff scan. |
| pendingCount | Gauge (Integer) | The number of scan splits assigned to readers but not yet fully processed. |

| Metric Name | Type | Description |
|---|---|---|
| unassignedCount | Gauge (Integer) | The number of scan splits waiting for assignment to a reader. |
| pendingRemovalCount | Gauge (Integer) | The number of completed splits deferred for removal until the next Flink checkpoint completes. |

# Monitoring replication job

Learn how to visualize the health of the replication pipeline by configuring Prometheus and Grafana.

To visualize the health of the replication pipeline, you must configure Prometheus and Grafana.

## Configuring Prometheus

1. Enable Prometheus metrics reporting in Flink by adding the following to the conf/config.yaml file:

```
metrics:
  reporter.prom.factory.class: org.apache.flink.metrics.prometheus.Prom
etheusReporterFactory
  reporter.prom.port: 9250-9260
```

2. Normalize Kudu tablet server metrics by using Prometheus metric relabeling and the json_exporter tool. This ensures metrics are stable and queryable by table name.
3. Use the provided scrape configurations to add a cluster label (such as source or sink) to all Kudu targets.

## Example Prometheus queries

Import the provided Grafana dashboard to monitor the following areas:

- **Job health:** Uptime, restart counts, and checkpoint durations.
- **Replication lag:** The difference between the current time and the lastEndTimestamp.
- **Write activity:** The rate of each write operation type on the source and sink clusters.
- **Throughput:** The number of records per second emitted by the source and consumed by the sink.

# Kudu tablet server metrics and table-level relabeling

Learn how to normalize Kudu tablet server metrics to enable table-level queries in Prometheus by using metric relabeling and the json_exporter tool.

## Metric normalization overview

The Kudu Prometheus endpoint (/metrics_prometheus) embeds the tablet ID directly into each metric name, such as kudu_tablet_<tablet-id>_rows_upserted . Because Prometheus requires stable metric names with labels for dimensions, you must use two additional components to make metrics queryable by table name:

- **Metric relabeling:** You must configure the Prometheus scrape configuration to extract the tablet ID into a tabl et_id label and normalize the metric name.
- **json_exporter:** This component reads the Kudu JSON metrics endpoint and exposes a kudu_tablet_info metric. This metric maps each tablet_id to its corresponding table_name, allowing dashboard queries to join on the ID and filter by table.

## Configuring json_exporter

To set up the mapping between tablet IDs and table names, you must install and configure the json_exporter tool.

1. Download the latest release from the prometheus-community/json_exporter page.

2. Place the binary file on a host that the Prometheus server can reach.
3. Create a configuration file named kudu_tablet_info.yml with the following content:

```
modules:
  default:
    metrics:
      - name: kudu_tablet
        type: object
        path: "{[?(@.type=='tablet')]}"
        labels:
          tablet_id: "{.id}"
          table_name: "{.attributes.table_name}"
          table_id: "{.attributes.table_id}"
        values:
          info: "1"
```

**Tip:** The json_exporter tool automatically appends _info to object-type metric names. The name: kudu_tablet property produces the kudu_tablet_info metric in Prometheus.

4. Start the json_exporter tool by running the following command:

```
./json_exporter --config.file=kudu_tablet_info.yml --web.listen-address=
:7979
```

## Prometheus scrape configuration for Kudu replication

Use the following Prometheus scrape configuration to collect metrics from Flink and Kudu clusters and to enable per-cluster query labeling.

### Configuring Prometheus scrape targets

The following configuration adds a cluster label, such as source or sink, to all Kudu tablet server targets. This labeling enables you to write per-cluster queries, such as comparing source write activity versus sink write activity, without using separate metric names or Prometheus jobs.

You must adapt the target addresses and label values to match your specific environment.

```
scrape_configs:
  # Flink replication job metrics.
  # JobManager exposes coordinator metrics (enumerator state, lastEndTimesta
mp) on port 9250.
  # TaskManagers expose operator-level metrics (records/sec, checkpoint in
fo) on port 9251+.
  - job_name: "replication"
    static_configs:
      - targets: ["<flink-jobmanager>:9250"]
        labels:
          app: "replication"
          component: "jobmanager"
      - targets: ["<flink-taskmanager>:9251"]
        labels:
          app: "replication"
          component: "taskmanager"
  # Kudu tablet server metrics.
  # metric_relabel_configs extracts the embedded tablet_id into a label
  # and normalizes the metric name so it is stable and queryable.
  - job_name: "kudu"
    metrics_path: "/metrics_prometheus"
```

```
    static_configs:
      - targets: ["<src-tserver-1>:8050", "<src-tserver-2>:8050"]
        labels:
          app: "kudu"
          cluster: "source"
      - targets: ["<snk-tserver-1>:8050", "<snk-tserver-2>:8050"]
        labels:
          app: "kudu"
          cluster: "sink"
    metric_relabel_configs:
      - source_labels: [__name__]
        regex: 'kudu_tablet_([a-f0-9]{32})_(.*)'
        target_label: tablet_id
        replacement: '$1'
      - source_labels: [__name__]
        regex: 'kudu_tablet_([a-f0-9]{32})_(.*)'
        target_label: __name__
        replacement: 'kudu_tablet_$2'
  # Kudu tablet -> table name mapping (by using json_exporter).
  # Scrape all tablet servers (source and sink) so the mapping covers both
clusters.
  - job_name: "kudu_tablet_info"
    metrics_path: /probe
    params:
      module: [default]
    static_configs:
      - targets:
          - "http://<src-tserver-1>:8050/metrics?types=tablet"
          - "http://<src-tserver-2>:8050/metrics?types=tablet"
          - "http://<snk-tserver-1>:8050/metrics?types=tablet"
          - "http://<snk-tserver-2>:8050/metrics?types=tablet"
    relabel_configs:
      - source_labels: [__address__]
        target_label: __param_target
      - source_labels: [__param_target]
        target_label: instance
      - target_label: __address__
        replacement: "<json-exporter-host>:7979"
```

**Tip:** To reload the scrape configuration without restarting Prometheus, start Prometheus with the --web.enable-lifecycle flag and run the following command:

```
curl -X POST http://localhost:9090/-/reload
```

## Grafana dashboard for Kudu replication

Learn about the ready-to-import Grafana dashboard for Kudu replication and how to use specific Prometheus queries to monitor Flink job health.

### Accessing the replication dashboard

A ready-to-import Grafana dashboard is available in the Kudu repository at the following location:

```
examples/flink-replication/monitoring/grafana/dashboards/replication.json
```

The queries described in this topic illustrate the data each panel displays and provide working examples from that dashboard. These examples assume you have configured the Prometheus scrape configuration with cluster labels on Kudu targets. You must adapt label selectors and metric names as needed for your specific environment, as other methods to express these queries exist.

### Monitoring job health

The primary goal of monitoring job health is to confirm that the Flink job is running, has not restarted unexpectedly, and is performing checkpoints normally.

#### Table 2: Prometheus queries for monitoring replication job health

| Goal | Prometheus query | Description |
|---|---|---|
| Job uptime | flink_jobmanager_job_uptime | Measures the time since the job last started or restarted. |
| Restart count | flink_jobmanager_job_fullRestarts | Tracks the number of full restarts. In a steady state, this value must be 0. |
| Checkpoint duration | flink_jobmanager_job_lastCheckpointDuration | Measures the duration of the most recent checkpoint. |

## Replication lag monitoring for Kudu

Learn how to monitor replication lag for Kudu by using Prometheus queries to verify that the enumerator is completing discovery cycles regularly.

### Understanding replication lag

The primary goal of monitoring replication lag is to verify that the enumerator is completing diff scan cycles at the expected intervals. In a steady state, the lag oscillates between 0 and approximately the value of the job.discoveryIntervalSeconds property divided by 60 minutes. This creates a sawtooth wave pattern in your monitoring tool.

If the lag value climbs continuously without resetting to zero, it indicates a stalled enumerator that requires investigation.

**Note:** On the first startup, the lastEndTimestamp value is 0 until the initial full snapshot scan completes. During this period, the lag query returns a very large value. This is the expected behavior of the system.

### Prometheus queries for replication lag

You can use the following queries to monitor replication lag in Grafana or Prometheus:

- Instantaneous lag: Use this query for a stat panel to see the current lag in minutes:

```
(time() - (flink_jobmanager_job_operator_coordinator_enumerator_lastEndTimestamp / 1000)) / 60
```

- Lag as a labeled time series: Use the timestamp() function so the result carries a label and can be plotted as a time series panel in Grafana:

```
(time() - (flink_jobmanager_job_operator_coordinator_enumerator_lastEndTimestamp / 1000)) / 60
```

- Lag as a labeled time series: Use the timestamp() function so the result carries a label and can be plotted as a time series panel in Grafana:

```
(timestamp(flink_jobmanager_job_operator_coordinator_enumerator_lastEndTimestamp)
- (flink_jobmanager_job_operator_coordinator_enumerator_lastEndTimestamp / 1000)) / 60
```

# Write activity monitoring for source and sink clusters

Learn how to monitor the rate of write operations on source and sink clusters by using Prometheus queries to verify data flow and pipeline drainage.

## Understanding Write Activity Metrics

Monitoring the rate of each write operation type independently for the source and sink tables helps you verify replication health.

- Source cluster: Use these metrics to verify that data is flowing into the system and to detect unexpected operation mixes.
- Sink cluster: Healthy replication primarily produces upsert operations. The presence of other operation types on the sink cluster is unexpected and requires investigation.

You also use these panels during the pre-stop checklist procedure to confirm that the replication pipeline has fully drained before you stop the job.

The cluster label, added during the Prometheus scrape configuration, separates source metrics from sink metrics without requiring separate metric names.

**Note:** The rate() function is preferred over the increase() function because it is resilient to counter resets caused by tablet server restarts or tablet leadership changes.

## Prometheus Queries for Write Activity

To create a clean legend in your monitoring tool, use one query per operation type. The following examples show queries for the source cluster. To build the equivalent sink panel, replace cluster="source" with cluster="sink".

- Source - insert rate (rows per minute):

```
sum(rate(kudu_tablet_rows_inserted{cluster="source"}[1m])
  * on(tablet_id) group_left(table_name)
  (max by (tablet_id, table_name, table_id) (kudu_tablet_info{table_name=
"<table-name>"}))) * 60
```

- Source - upsert rate (rows per minute):

```
sum(rate(kudu_tablet_rows_upserted{cluster="source"}[1m])
  * on(tablet_id) group_left(table_name)
  (max by (tablet_id, table_name, table_id) (kudu_tablet_info{table_name=
"<table-name>"}))) * 60
```

- Source - update rate (rows per minute):

```
sum(rate(kudu_tablet_rows_updated{cluster="source"}[1m])
  * on(tablet_id) group_left(table_name)
  (max by (tablet_id, table_name, table_id) (kudu_tablet_info{table_name="
<table-name>"}))) * 60
```

- Source - delete rate (rows per minute):

```
sum(rate(kudu_tablet_rows_deleted{cluster="source"}[1m])
  * on(tablet_id) group_left(table_name)
  (max by (tablet_id, table_name, table_id) (kudu_tablet_info{table_name="
<table-name>"})
```

# Enumerator split state monitoring for Kudu replication

Learn how to monitor the state of scan splits during a discovery cycle by using Prometheus queries to identify stalled readers.

## Understanding split state metrics

The primary goal of monitoring the enumerator split state is to track the progress of scan splits through a discovery cycle. In a steady state between cycles, the values for all three split metrics must be zero.

A non-zero value for the unassignedCount or pendingCount metric that does not drain between cycles indicates a stalled reader that requires troubleshooting.

During an active discovery cycle, the following behaviors are expected:

- The unassignedCount metric spikes to the number of tablets and drains to zero as the job assigns splits to readers.
- The pendingRemovalCount metric rises briefly and drops to zero after the next Flink checkpoint completes.

## Prometheus query for split states

To visualize these states in a single Grafana panel, use the label_replace function to unify the metrics. You must set the Grafana legend to {{split_state}}.

Run the following query to monitor the split states:

```
label_replace(flink_jobmanager_job_operator_coordinator_enumerator_pendingC
ount,
              "split_state", "pending", "", "")
or
label_replace(flink_jobmanager_job_operator_coordinator_enumerator_pendingR
emovalCount,
              "split_state", "pendingRemoval", "", "")
or
label_replace(flink_jobmanager_job_operator_coordinator_enumerator_unassi
gnedCount,
              "split_state", "unassigned", "", "")
```

# Monitoring throughput and row count consistency for Kudu replication

Learn how to use Flink and Kudu metrics to verify record flow, detect backpressure, and perform coarse consistency checks between source and sink clusters.

## Flink operator throughput

The primary goal of monitoring operator throughput is to confirm that records flow through the pipeline at the expected rate. The KuduSource output rate and the KuduSink input rate must track each other closely. A growing gap between these two metrics indicates backpressure or a slow writer.

Use the following Prometheus queries to validate the record flow:

**Records per second emitted by KuduSource:**

```
sum(flink_taskmanager_job_task_operator_numRecordsOutPerSecond{o
perator_name="Source:_KuduSource"})
```

**Records per second consumed by KuduSink:**

```
sum(flink_taskmanager_job_task_operator_numRecordsInPerSecond{op
erator_name="Sink:_Writer"})
```

### Live row count consistency check

You can perform a coarse consistency check by comparing the approximate total row count on the source and sink tables. The sink row count must track the source count within one discovery interval. When you apply the cluster label in the scrape configuration, both clusters appear as separate series in a single query.

Run the following query to compare row counts:

```
sum(
  kudu_tablet_live_row_count
  * on(tablet_id) group_left(table_name)
  (max by (tablet_id, table_name) (kudu_tablet_info{table_name="<table-na
me>"}))
) by (cluster)
```

You must set the Grafana legend to {{cluster}}.

**Important:** Row counts do not reflect upsert operations. If the sink already contains a key that the source cluster upserts, the sink count remains unchanged even though the row content changed. Row count equality is a necessary but not sufficient condition for data consistency.

### Monitoring multiple tables

Each replicated table requires a dedicated replication job. When you monitor multiple tables, you must update the tabl e_name filter in the Grafana queries to match the specific table.

The kudu_tablet_info mapping metric covers all tables on the tablet server automatically. You do not need to make changes to the json_exporter or Prometheus configuration when you add new tables.

## Troubleshooting Kudu Replication

Learn where to find error logs, how to resolve common replication errors, and how to identify the cause of a stalled replication job.

### Error log locations

The replication job embeds the Kudu Java client directly inside the Flink TaskManagers and JobManager. Kudu-level errors, such as scan failures or write errors, appear as standard Java exceptions in the Flink logs rather than in the Kudu master or tablet server logs.

Check the following locations for specific errors:

- **TaskManager logs:** These logs contain errors that occur during data reading (scan RPCs) or writing (upsert or delete RPCs).
- **JobManager logs:** These logs contain errors related to the enumerator, such as split discovery and diff scan scheduling, as well as checkpoint coordination and job-level failures.

To access these logs, you can use the YARN Resource Manager user interface (UI) or run the following command in the command-line interface (CLI):

```
yarn logs -applicationId <yarn-application-id>
```

### Common errors and remediation reference

The following table describes common error messages and their respective solutions:

| Error message | Cause | Remediation |
|---|---|---|
| Snapshot too old | The Multi-Version Concurrency Control (MVCC) history retention window was exceeded. | Stop the job, delete the checkpoint directory, and restart without the -s flag. Increase the --tablet_history_max_age_sec property. |
| Not found: The table does not exist | The sink table was not created and job.createTable is set to false. | Create the sink table manually or restart the job with --job.createTable true. |
| Timed out: deadline exceeded (writes) | The sink cluster is overloaded or unreachable. | Check the health of the sink tablet server. Increase the writer.operationTimeout property. |
| Timed out: deadline exceeded (scans) | The source cluster is slow or overloaded. | Check the health of the source tablet server. Increase the reader.scanRequestTimeout property. |
| Authentication error or GSSAPI failures | The Kerberos ticket expired or is unavailable. | Renew the ticket and resubmit the job, or use Flink keytab-based authentication. |
| ClassCastException | There is a conflict between the Kudu and Hadoop Kerberos classloaders. | Add the -Dclassloader.parent-first-patterns.additional=org.apache.kudu argument to the submission command. |

### Investigating a stalled replication job

If the lastEndTimestamp metric stops increasing, the enumerator is not completing discovery cycles. To resolve this, perform the following actions:

1. Check the TaskManager logs for scan timeout errors or hanging RPCs. A non-zero pendingCount metric that does not drain indicates a stuck reader.
2. Check the JobManager logs for checkpoint failure messages, such as Checkpoint expired before completing.
3. Review the Flink Web UI for repeated Job restarting entries and identify the root exception in the TaskManager logs.

### Resolving startup failures

If the job fails immediately after submission, you must verify the following configurations:

1. Ensure that the job.checkpointingIntervalMillis property is strictly less than the product of job.discoveryIntervalSeconds and 1000.
2. Verify that the checkpoint directory path is accessible and exists.
3. Confirm that the source and sink master addresses are correct and that the masters are reachable.