Chapter 4. Kudu Administration

The administrator's dream is to work with a system that is resilient, fault tolerant, simple to scale, monitor, and adapt as requirements change. Even better is a system that is self-healing, shows warning signs early, and lets the administrator go to sleep knowing that the system will run in a *predictable* way.

Kudu is designed and built from the ground up to be fault tolerant, scalable, and resilient and provides administrators with the means to see what's happening in the system, both through available APIs and visually through a web user interface (UI) and a handy command-line interface (CLI).

In this chapter, we get you started as an administrator so that you can hit the ground running. Installation options are covered in previous chapters, so we begin here by looking at planning for your Kudu deployment and then walk through some of the most common and useful administrator tasks.

Planning for Kudu

Let's begin by taking time to understand how to appropriately plan for a Kudu deployment.

and more provided at *http://kudu.apache.org/*. However, Kudu might also be included in various Hadoop distributions. If going with a distribution, always refer to the documentation provided by the vendor as deployment strategies may vary.

Kudu can be installed as its very own cluster with no dependency on any other components; however, we expect that many installations will be on existing Hadoop clusters. Knowing how to plan, size, and place Kudu services in relation to all your other Hadoop components is also be covered.

Master and Tablet Servers

Before we jump into hardware planning and where we will place the Kudu services, as an administrator we need to understand the primary two components we need to think about in Kudu: the master and tablet servers. These servers manage tables, or rather, *tablets*, that make up the contents of a table. Tablets are then replicated, which is why they are also known as replicas.

Typically, you start a cluster with three master servers. Consensus needs to be reached between these master servers so that they agree among themselves which server is the leader. This decision-making process must have a majority of servers "vote" to make the final decision (e.g., 2 of the 3, or 3 of the 5, and so on), and the decision they make is final. This procedure is both fault tolerant and performant, using what's called the Raft consensus algorithm, which is a variation of the well-known Paxos consensus algorithm. Three or five master servers are good examples of the number of master servers to plan for, whereas seven master servers would be considered excessive. The number *must* be odd, where you can lose up to $(n - 1) / 2$ servers and still be able to make progress. In the event that *more* than $(n - 1) / 2$ servers fail, the master service will no longer be making progress and thus is no longer available to the cluster.

Tables break down their data into partitions stored in this concept of "tablets," which are then replicated where the default replication fact i referred to as a *replica* that contains a portion of data that m

This metadata is currently stored in a table that resides on the master servers, which has exactly one tablet stored on the master servers.

Let's say you decide to create table T, which we depict with the logical box illustrated in Figure 4-1. Data for this table is broken down into chunks called tablets, which we show as having three tablets: Tablet 1, Tablet 2, and Tablet 3. Table T also stores your inserted data.

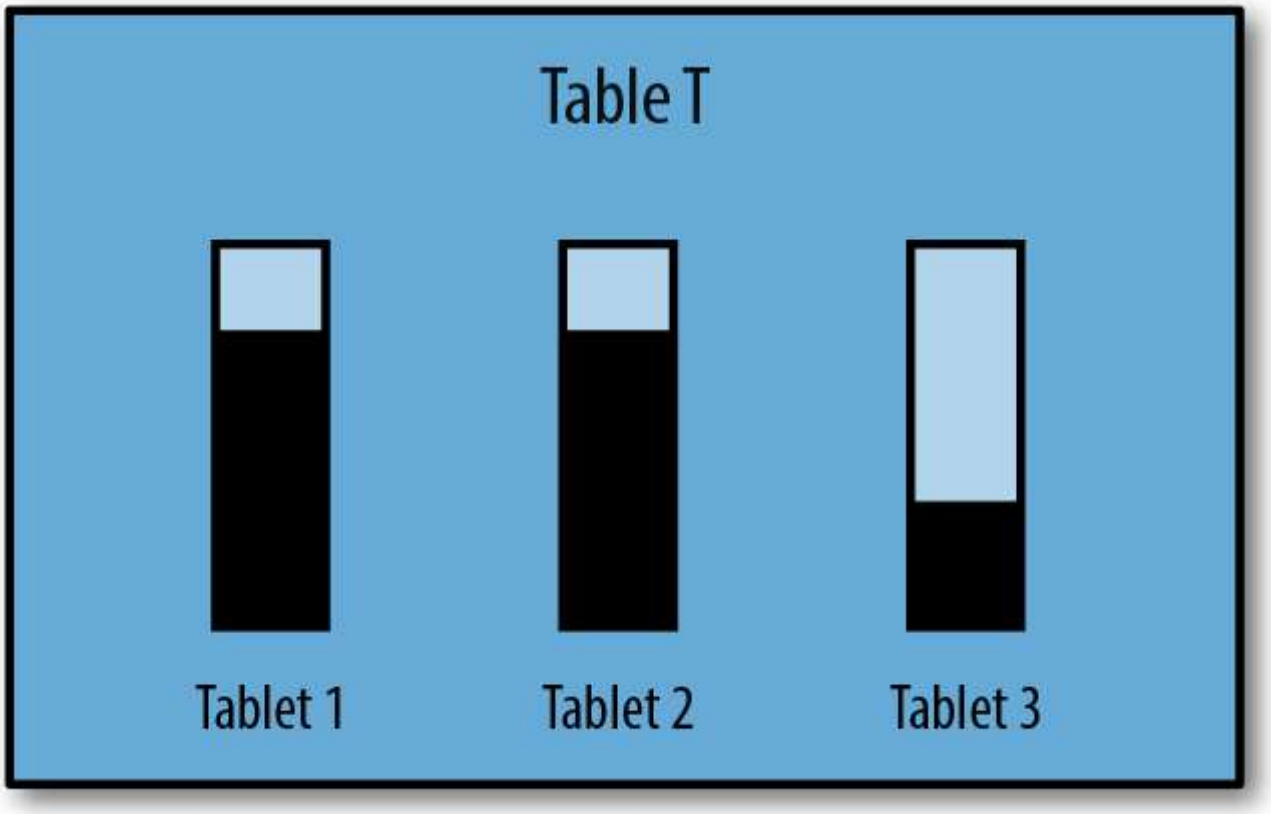


Figure 4-1. Table T logical representation

This representation of the table, broken down into tablets, is spread further across nodes by having each of these tablets replicated across several tablet servers. In this example of three tablet servers (Figure 4-2), we mark each tablet as 1, 2, 3, but because there are multiple replicas, one of these replicas is the leader (denoted by L) whereas the other two are followers (denoted by F).

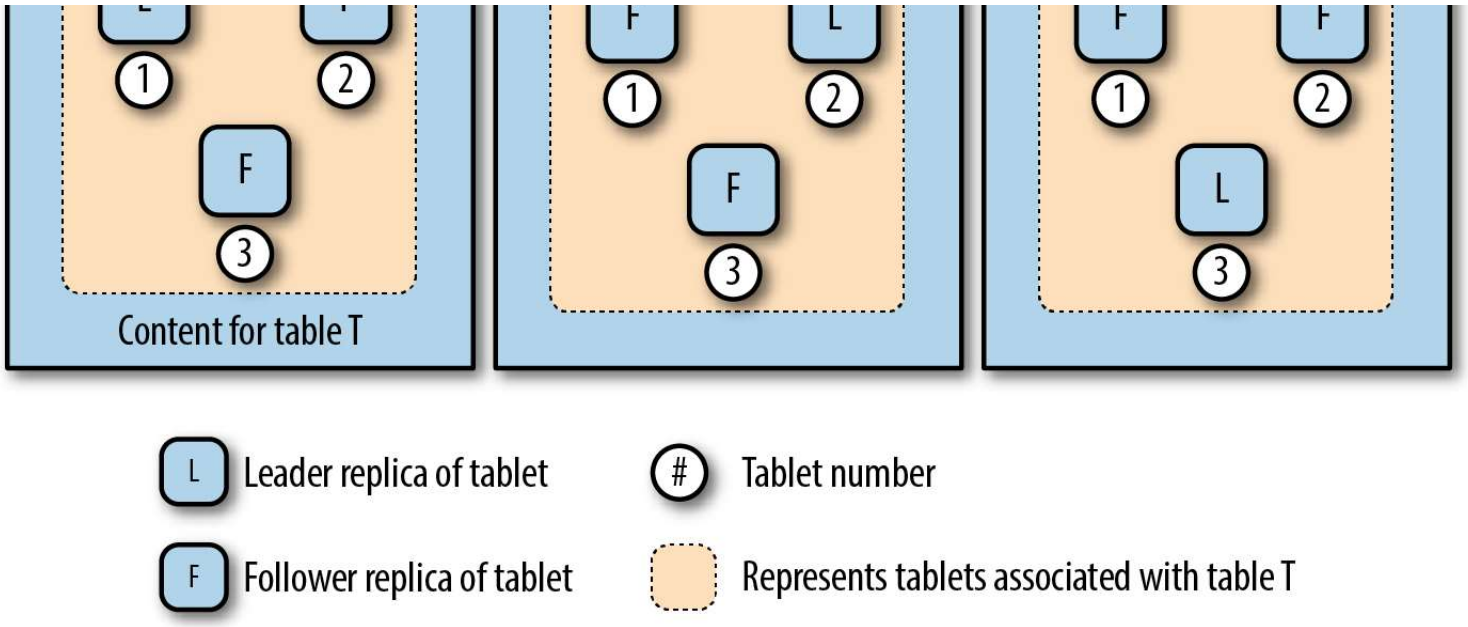


Figure 4-2. Table T replicated

Raft consensus is used to elect a leader among the tablet replicas on the tablet servers.

Master tablets, because they contain only metadata, really should not be as hot as your user tablets given that we could expect that you're not performing Data Definition Language (DDL) operations as fast as you are loading your data into tablets. Likewise, these tables should never grow as large as your user tables, so the requirements for the master servers for storage, memory, and CPU will be a lot more modest than your tablet servers themselves.

The system catalog table is defined as a three column table consisting of `entry_type`, `entry_id`, and `metadata`. The catalog manager loads this table into memory and constructs three hashmaps to allow fast lookups of the catalog information. The catalog information is compact, and since it contains only metadata and no real user data, it will remain small and not require large amounts of resources such as memory and CPU.

TIP

Master servers contain only metadata about your user tables. Therefore, storage, memory, and compute resource requirements are considerably smaller than tablet servers. Sizing details are discussed later in the chapter.

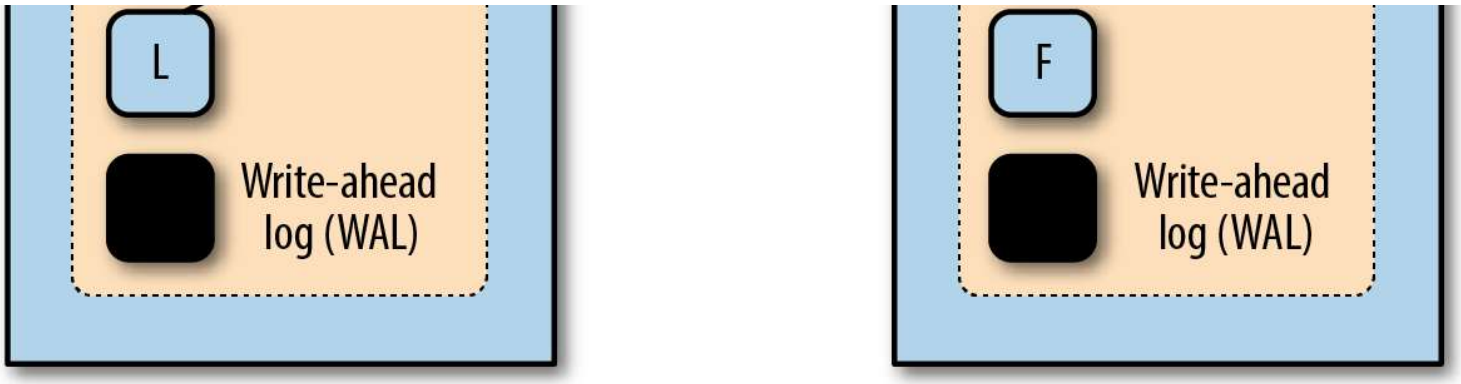
elected the leader, whereas the others are followers providing high availability for this data. Finally, we have a write-ahead log existing on each of the master servers.

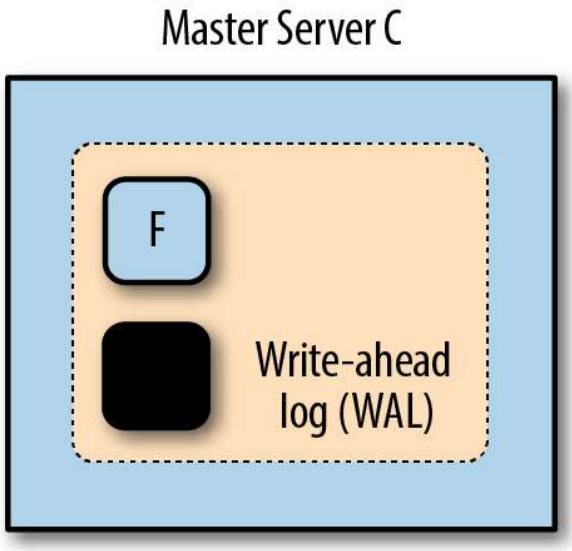
On the lower half of the diagram, we have tablet servers that manage the tables you create in Kudu. We have *N* tablet servers, and we depict tables with the dotted line. Table data is stored in tablets that are spread across the various tablet servers. Once again we see one tablet is selected to be the leader, denoted by L, while the followers, denoted by F, for a given tablet replica are found on other tablet servers. We also show that for each Kudu table, a separate write-ahead log (shown in Figure 4-3 as "WAL") is created and exists on each tablet server.

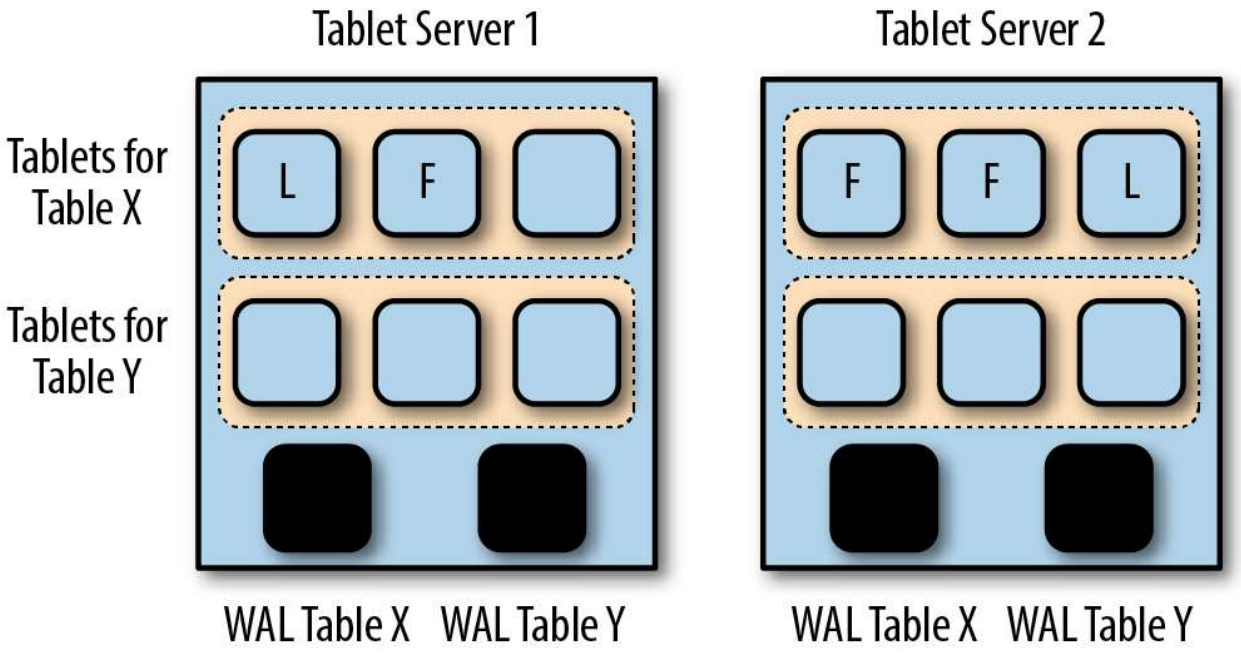
In fact, master servers and tablet servers are identical at this high-level view, showing the idea that the same concepts exist across both of these servers.

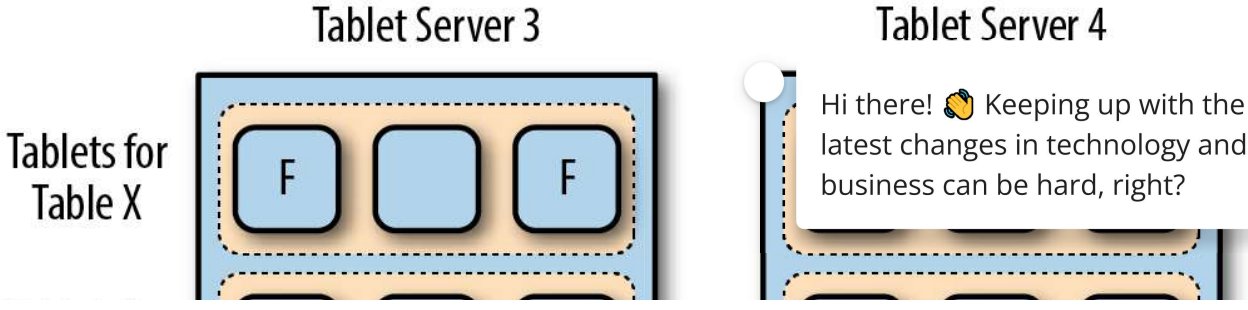
How many tablet servers should you plan for? Of course, the answer is going to be *it depends*. Kudu stores data in columnar, Parquet-like format, using various encodings and compression strategies. Because it is *Parquet-like*, from a storage *capacity* perspective, you can roughly examine how much capacity your data occupies in Parquet format files in your HDFS filesystem (or otherwise) and use that as a rough measuring stick for how your data would look stored in Kudu.

Traditional Hadoop-related technologies were not built from the ground up to effectively use solid-state drives (SSDs) because SSDs were too expensive. Kudu being a modern platform, takes advantage of SSD performance, and can make use of NVM Express (NVMe), extremely fast PCIe-attached flash adapters. SSD drives are still not quite as dense as hard-disk drives (HDDs) today and this means that typically these hardware configurations would have less overall storage density per node than HDD configurations.









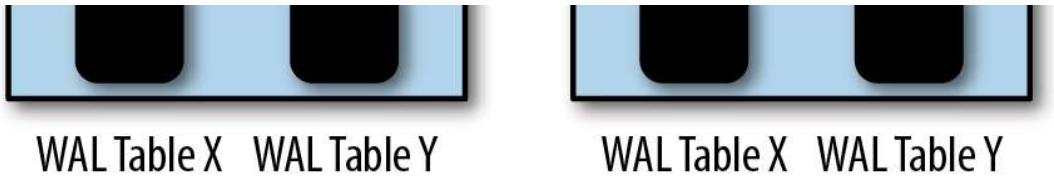


Figure 4-3. High-level architecture

That being said, as it stands today, a reasonable *default* configuration would plan for a range of less than 10 TB of data on disk for a given tablet server (includes storage of all tablet replicas) and a maximum range of low thousands of replicas per tablet server. These numbers are on the cautious end of the spectrum, and as Kudu matures, more data stored on a tablet server will be perfectly acceptable. From a tablet server count perspective, the following formula might help your initial sizing:

```
d = 120 TB : Size of dataset in parquet format
k =   8 TB : Target max disk capacity per tablet server
p =    25% : Percentage overhead to leave
r =      3 : Tablet replication factor
```

Now from a capacity view, we can do the following simple calculation, to determine the number of tablet servers we need for our dataset:

```
t = (d / (k * (1 - p))) * r
t = (120 / (8 * (1 - 0.25))) * 3
t = 20 tablet servers
```

There are more considerations to take into account, such as planning for how many replicas would exist on a tablet server to ensure that we are within the low thousands range. Those calculations depend on partitioning schema strategy as well as the number of tables expected to be defined in your environment. We leave it to you to review Chapter 6, which specifically goes into schema design, and then it's a matter of simple math to ensure that you meet these best-practice ranges.

Write-Ahead Log

related to preallocated storage, perhaps asynchronously. They have a compression codec as well as minimum and maximum number of segments that should be kept around so that other peers might be able to catch up if they've fallen behind.

The WAL is an append-only log that at first glance would look like the disk just needs to be able to perform high sequential writes. Even though these are append-only logs, there are logs for each tablet on the master or tablet server. If multiple tables are being written to, from the disk's point of view, it is very much a random write pattern.

If we look at performance characteristics of HDDs versus SSDs from a random write workload perspective where we're interested in a high amount of low-latency I/O operations (as opposed to high throughout sequential write operations), SSDs have an enormous advantage. Where HDDs will have I/O operations per second (IOPS) on the order of the low hundreds, SSDs will have IOPS in the thousands to tens of thousands of operations. In 2016, Samsung showcased a million IOPS SSD, which is certainly orders of magnitude faster than HDDs! Just looking at write IOPS alone, it is in the low hundreds of thousands of IOPS; needless to say, these are drastic differences. Read/write latency is below 100 microseconds, whereas HDD is in the 10 millisecond range. You get the idea.

Given the type of workload, it is best to plan on a fast SSD NVMe solution for the WAL.

Capacity-wise, default log segment sizes are 8 MB with a minimum of 1 log segment kept and a maximum of 80. This is not a hard maximum, because there are some cases in which a leader tablet might continue to accept writes while the replicas on other peers are restarting or are down.

Kudu might improve the disk usage of the WAL with various techniques in future releases, though we can still plan around these worst-case scenarios. If all tablets are writing at the same time, and staying within the current recommended range of the low thousands, say, 2,000 tablets per tablet server, the math is simple:

```
8 MB/segment * 80 max segments * 2000 ta                    .3 TB
```

Typical 2U servers used as worker nodes in a big data environment will have 2× SSD in the back of the unit for the operating system (OS), and either 12 x 3.5″ LFF or 24 x 2.5″ SFF drive bays in the front. There are a few options for how to prepare storage for the WAL:

- Install dedicated SSDs in a drive bay on the front
- Make one of the two SSDs in the back dedicated to the WAL (losing out on Redundant Array of Independent Disks [RAID] protections for the OS)
- Create a physical or logical partition on the pair of rear SSDs, and dedicate a mount point to the Kudu WAL
- Install an NVMe PCIe SSD interface–based flash drive

For large production deployments, we do not recommend setting the WAL to a dedicated HDD, or worse a shared HDD with other services, because it will affect both write performance and recovery times in failure scenarios.

Performance of the various storage options vary incredibly. Table 4-1 summarizes them for your consideration.

Table 4-1. HDD versus SSD versus NVMe PCIe flash storage

Storage medium	IOPS	Throughput (MB/sec)
HDD	55-180	50-180
SSD	3,000-40,000	300-2000 (SAS max is 2,812 MB/sec)
NVMe PCIe flash storage	150,000 up to 1 million+	

which greatly benefits Kudu workloads.

Data Servers and Storage

When it comes to your data—what we call user data—this is where it becomes simple. Provide as many available HDDs (or better yet SSDs!) on your server as possible for storage. You can scale out your servers, at will, though note that data is not, as of yet, automatically distributed across new servers added to the cluster.

It is highly likely that you will want to enable Kudu in an HDFS-backed big data cluster. You can configure Kudu storage directories on the very same HDFS-specific mount points. For example, if you have a disk that is partitioned and formatted with a filesystem, mounted on `/disk1` you might likely have a directory called `/disk1/dfs` representing the DataNode of your distributed filesystem, HDFS, installation. Go ahead and create a directory called `/disk1/tserver` for your Kudu content.

Some consider whether to dedicate a few disks on a server just for Kudu and then others for HDFS. Although this is possible and can even in fact yield a performance improvement, it is not advisable to go through the trouble of overhead in management, dealing with expansion and checking which disks are really being used effectively.

HDFS is certainly aware of how much capacity is still available per disk, and in 2016, it even introduced the ability to rebalance data blocks within a DataNode (HDFS-1312: Rebalance disks within a DataNode) according to either round-robin or available space policies to ensure usage is even.

NOTE

At the time of this writing, it is recommended to not store more than 8 TB of Kudu data per tablet server. This is after considering replication as well as compression in well-packed columnar storage formats. Density per node is sure to improve as the product continues to gain adoption and maturity, though it is currently an important point to consider, especially with regard to worrying about sharing disks with otl

latency, such as an SSD. In previous releases to v1.7, this metadata is written to the first entry listing the set of data directories on a tablet server. From v1.7 and onward, the default directory is now the directory specified by the WAL directory. However, there is a new parameter, `--fs_metadata_dir`, that allows you to take control and specify where the metadata should be placed.

Placing the metadata on the WAL or nondata disk is helpful because this data can grow over time, and your data drives can become skewed in how they are populated.

Replication Strategies

As you define your replication factor for a given table, tablet servers will strive to ensure that that replication factor is preserved for all the tablets within that table.

Hence, if a tablet server goes down, the number of replicas might have dropped from three replicas to two, Kudu will look to heal these tablets quickly.

The replication strategy primarily is use is named *3-4-3*, which is intended to say that if a tablet server goes down, before evicting that failed replica, Kudu will first add a replacement replica and then decide it is time to evict the failed one.

Another strategy, named *3-2-3*, would evict that failed replica immediately and then proceed to add a new one. However, for systems that might go offline only periodically and come back up, it causes a much longer delay in becoming part of the cluster again.

In environments that might have more frequent server failures, this becomes important for overall stability. Otherwise you would have a situation in which the failed tablet server has all its tablets evicted immediately, leading to a lot of work to bring it back into play. This actually allows for very fast recovery when a tablet server goes down only briefly and then returns.

There are situations in which Kudu will perform the 3-2-3 mechanism, but only if the failure that one of the tablet servers experiences is known t , a n tablet replica will be created on a new host, and if th back to life, no harm done, and the new copy being created is simply canceled.

- A brand-new Kudu-only cluster
- A brand-new Hadoop cluster that inclues Kudu
- An existing Hadoop cluster to which we add Kudu

We cover considerations for each of these scenarios.

New Kudu-Only Cluster

When embarking on creating a brand-new cluster targeted for Kudu workloads, in an ideal world, servers would be ordered with all data drives being SSDs, plus an NVMe PCIe SSD interface–based flash drive for the WAL.

Actually, this is not far-fetched for a Kudu-only cluster. Considering that the overall recommended storage density for Kudu today should be on the order of approximately 10 TB, it is easy to accommodate a purchase of a typical 1U server, with two SSDs in the back for the OS and eight SSDs for data coming in around the 1.8 TB mark, which would yield 14.4 TB of raw capacity per server. Add to that a 1 TB NVMe PCIe attached flash drive, and you have a very capable cluster meeting the design goals of Kudu (Figure 4-4).

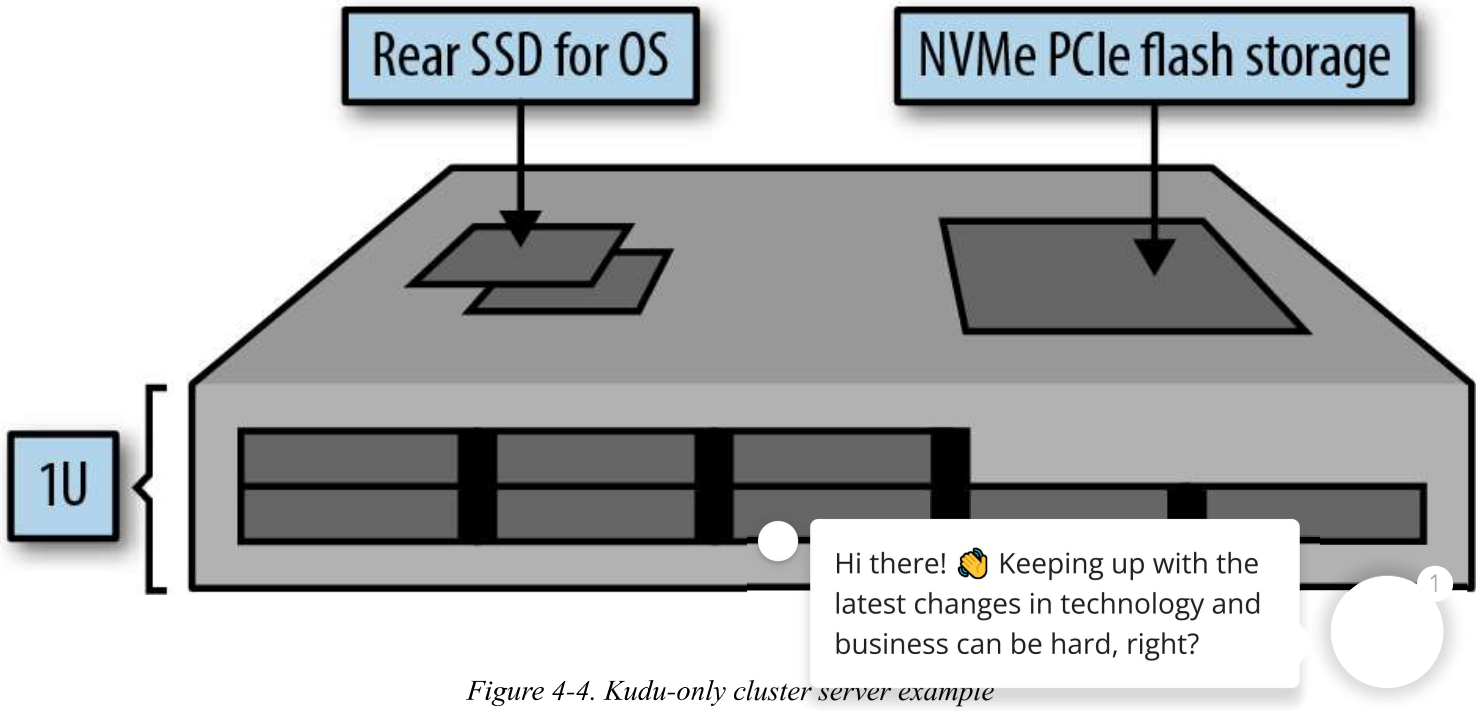


Figure 4-4. Kudu-only cluster server example

it to be part of a larger Hadoop ecosystem that would include HDFS, along with the various other processing services on top, such as Hive, Impala, HBase, YARN, Solr, and Spark.

In this case, Kudu integrates right into the rest of the ecosystem seamlessly, and for write-heavy workloads in particular, we want to ensure the placement of the WAL is on the fastest medium we could include.

We propose two different approaches for planning in this kind of environment: one in which the WAL would be on NVMe PCIe–attached flash storage, and the second using one of the data drives dedicated to the WAL.

Starting with the three master servers in a typical Kudu deployment, this fits in nicely with the master servers used in a common Hadoop environment.

Typically in Hadoop, we will have nodes with "master" type services, such as the following:

- Active/Standby NameNode for HDFS High Availability, where the JournalNodes are spread across three master servers leveraging dedicated spindles
- ZooKeeper instances spread across three master servers, provided with dedicated spindles
- YARN Active/Standby ResourceManager plus the YARN Job History Server
- One or more HiveServer2 instances (though these can be on edge nodes, as well)
- One or more HBase master servers
- Spark Job History Server
- Impala StateStore and CatalogStore
- Sentry Active/Standby nodes
- Multiple Oozie Services

Hence, it is a natural fit for Kudu to require a fast drive for the WAL even here, similar to JournalNodes and ZooKeeper with their dedicated drives. We want Kudu to have its own fast drive, as well.

Again, we reuse the idea of using a typical 1U server with eight drives out front, and suggest that all of these be SSD drives. We rely on a NVMe PCIe flash storage adapter for the Kudu master WAL, and accommodate dedicated drives (a pair of drives in RAID1) for various services like the NameNode, JournalNode, and ZooKeeper services (Figure 4-5).

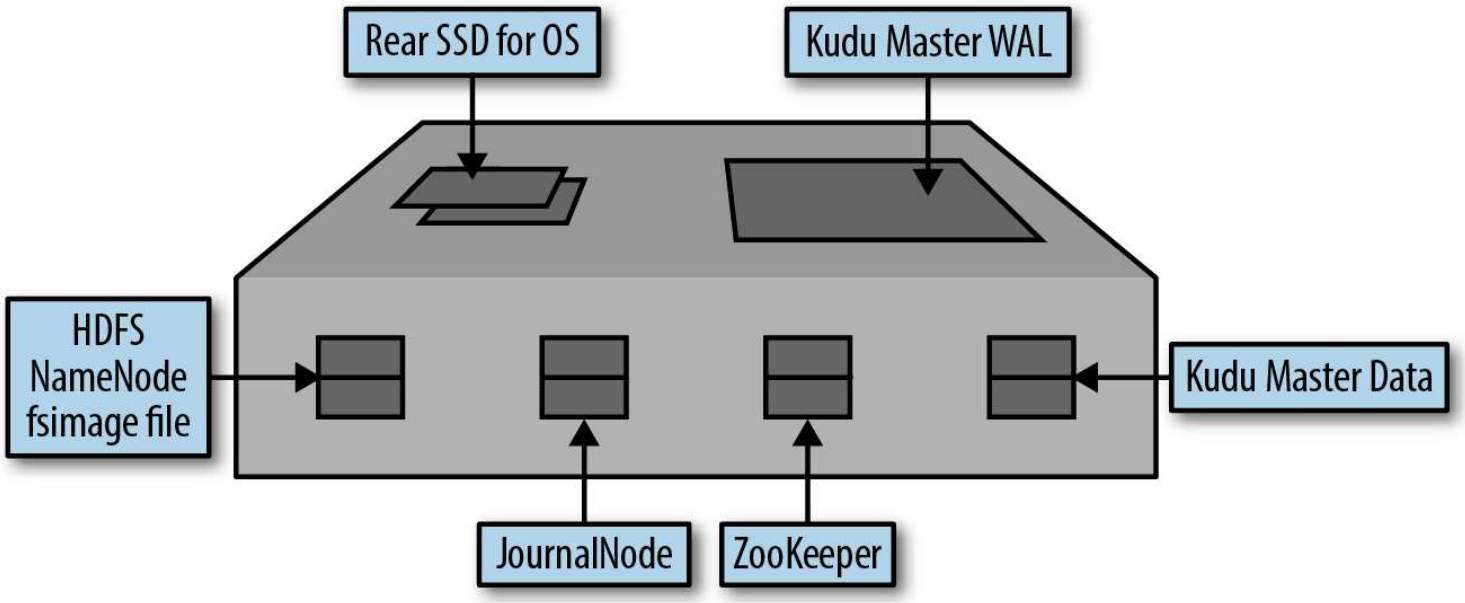


Figure 4-5. Master server for Hadoop with Kudu

The second option we offer here is to consider using a typical 2U server that has two SSDs in the back for the OS and twelve 3.5″ drives in the front. With each pair of drives in RAID1 configuration, we end up with six block devices made available to the OS. A filesystem is then formatted and mounted on each of these block devices, where we lay out each filesystem to be dedicated to the following:

- NameNode fsimage files
- Hive Metastore database (typically MySQL or Postgres)
- HDFS JournalNode

- Kudu master WAL

Only the last four drives for Kudu would be optimized with SSDs, but the rest of the services would benefit from SSDs as well. In this example, we do not configure a NVMe PCIe–attached flash storage device in an attempt to fully utilize the drive bays available in 2U units while still providing more than enough capability for Kudu (Figure 4-6).

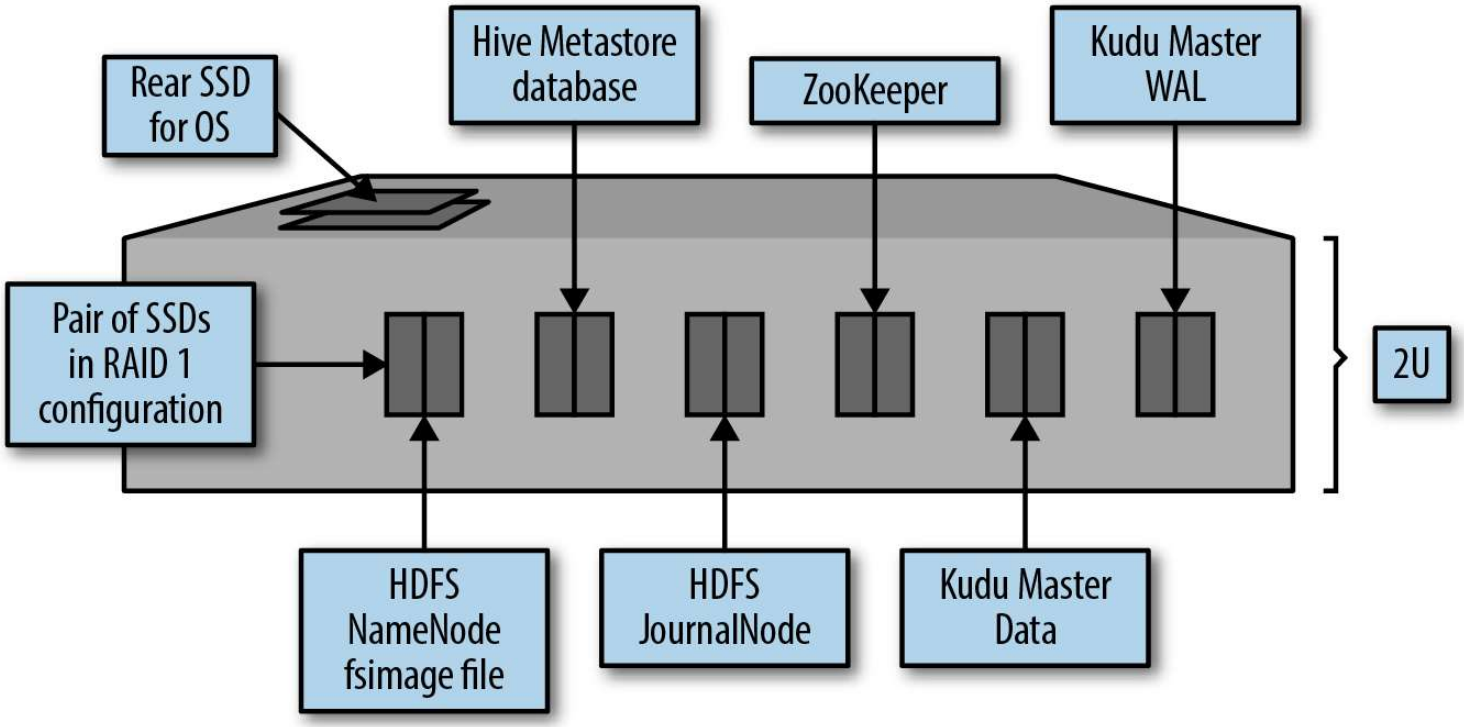


Figure 4-6. Master server for Hadoop with Kudu, option 2

The next consideration is planning for the Kudu tablet servers amidst the Hadoop ecosystem, namely Kudu data on each of the HDFS DataNodes. These are the scale-out servers that are used for a variety of other services in the Hadoop ecosystem, and we refer to them as *worker nodes* as a more generic term.

Worker nodes are most often represented in the Hadoop space as 2U, industry-standard servers with either twelve 3.5″ drive bays in the front or twenty-four 2.5″ drive bays in the front for data. There are typically two 2.5″ drive [illegible] ant for the OS.

ally be all SSDs to benefit Kudu. However, it is understandable and still expected that perhaps even for newly provisioned environments, HDDs would still be commonplace. They still provide much more density per server, which may be an important requirement, especially for data sitting in HDFS.

As soon as users learn that Kudu sits completely outside of HDFS and the rest of the Hadoop ecosystem, the common thinking is to start isolating drives for Kudu versus HDFS. Although this is possible and certainly does isolate workloads from a storage perspective, it can end up being too restrictive in the long run. There might be workloads today that require more HDFS capacity; however, in the future, perhaps more of those workloads will move to Kudu. Making changes in storage configurations after the fail could prove to be very costly and cumbersome.

We would suggest that you use the same disks assigned for HDFS data for Kudu data as well, except for the actual directory path given to each service.

For example, let's assume that `/disk1` is an `ext4` filesystem that sits on top of a single `JBOD` disk. For HDFS, normally, you would define a path under this directory such as:

```
/disk1/dfs : Directory for HDFS data
```

For Kudu, we simply would add a new directory such as the following:

```
/disk1/tserver : Directory for Kudu data
```

Hence, both Kudu and HDFS sit in their own directories on the same filesystem and device volume. HDFS and Kudu alike would simply know about the total capacity left on a given drive, and an HDFS rebalance might take place, for example, if Kudu data is hotspotting much of its data on this one node.

The one situation for which we would recommend [illegible] encryption at rest. You should configure HDFS with HDFS Tra[illegible]

If you select servers with all SSD drives, both HDFS and Kudu would benefit. If all the drives are HDD, HDFS and Kudu would use them, though Kudu would likely be a little more affected in its ability to read/write. Still, this is expected and commonplace, and many workloads already run with this behavior so it should not be considered a deterrent.

The 3.5″ HDD drives today can easily be in the range of 6 to 8 TB in capacity, and reserving one or a pair of these for the Kudu WAL would result in a lot of wasted capacity given to the WAL instead of to data. Thus, we certainly suggest NVMe PCIe–attached flash storage. This kind of approach would yield the hardware topology as shown in Figure 4-7.

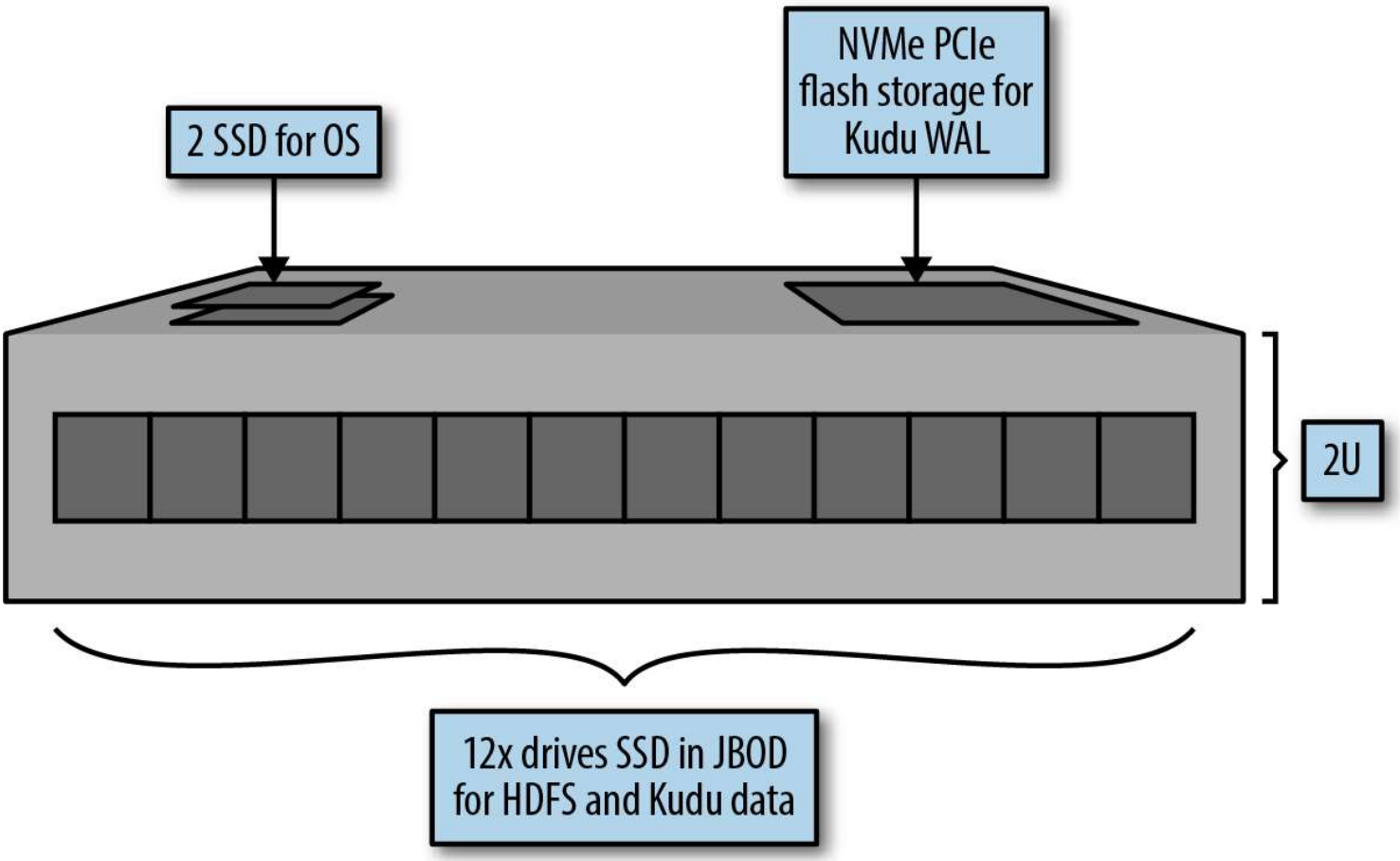


Figure 4-7. Tablet server for Hadoop with Kudu, option 1

Another consideration is a topology without using NVMe PCIe–attached storage. In this case, to ensure we don't waste as much space for the WAL, we take a 2U server with twenty-four 2.5″ drive bays. We take just a single drive bay to dedicate to the Kudu WAL, and ideally, we could make this one drive SSD (if th

This is better depicted by Figure 4-8.

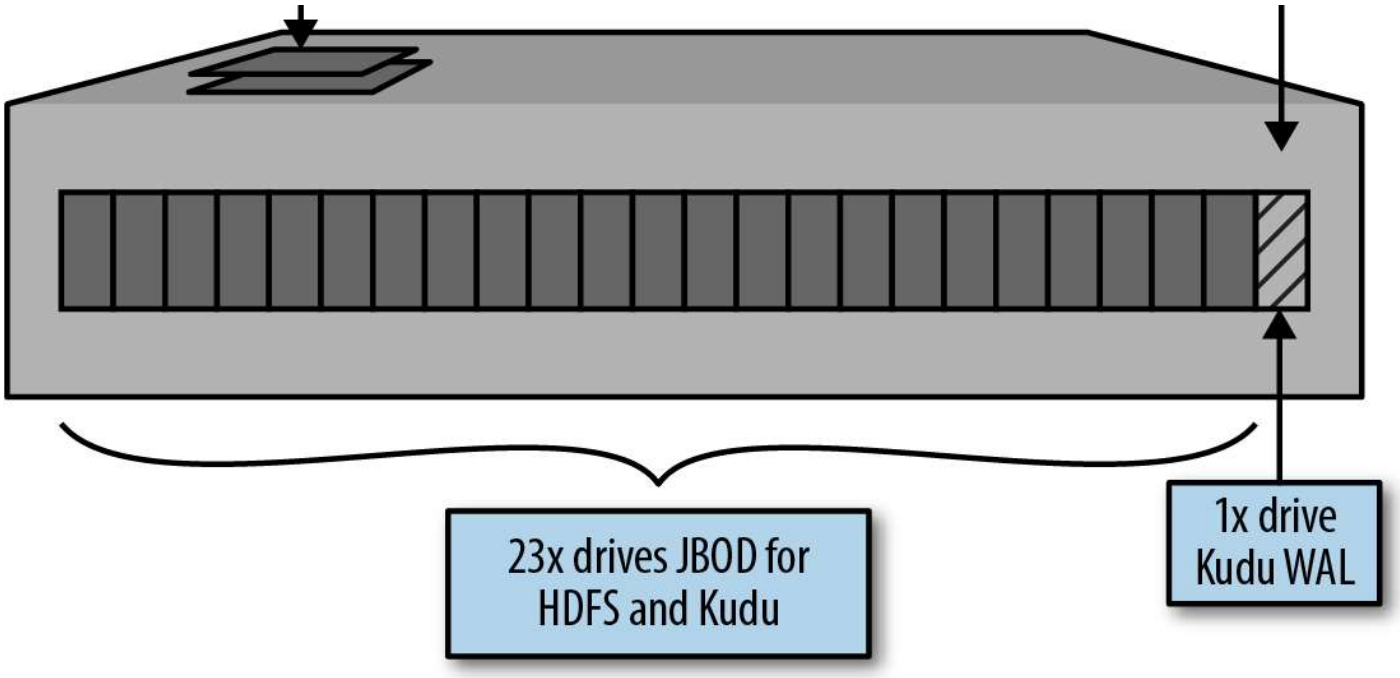


Figure 4-8. Tablet server for Hadoop with Kudu, option 2

With this strategy, if you ever remove Kudu from this cluster, that disk drive dedicated to the WAL can quickly be repurposed for HDFS, for example. So it gives some flexibility as well.

Add Kudu to Existing Hadoop Cluster

The previous section went through a discussion about optimal configuration of Kudu in a brand-new Hadoop environment. Our goal, obviously, is to get as close as possible to those topologies.

The quickest way to get to an optimal Kudu configuration is to order and install a new NVMe PCIe flash storage device. By doing so, you get closer to the configuration as depicted in an earlier diagram, shown in Figure 4-9.

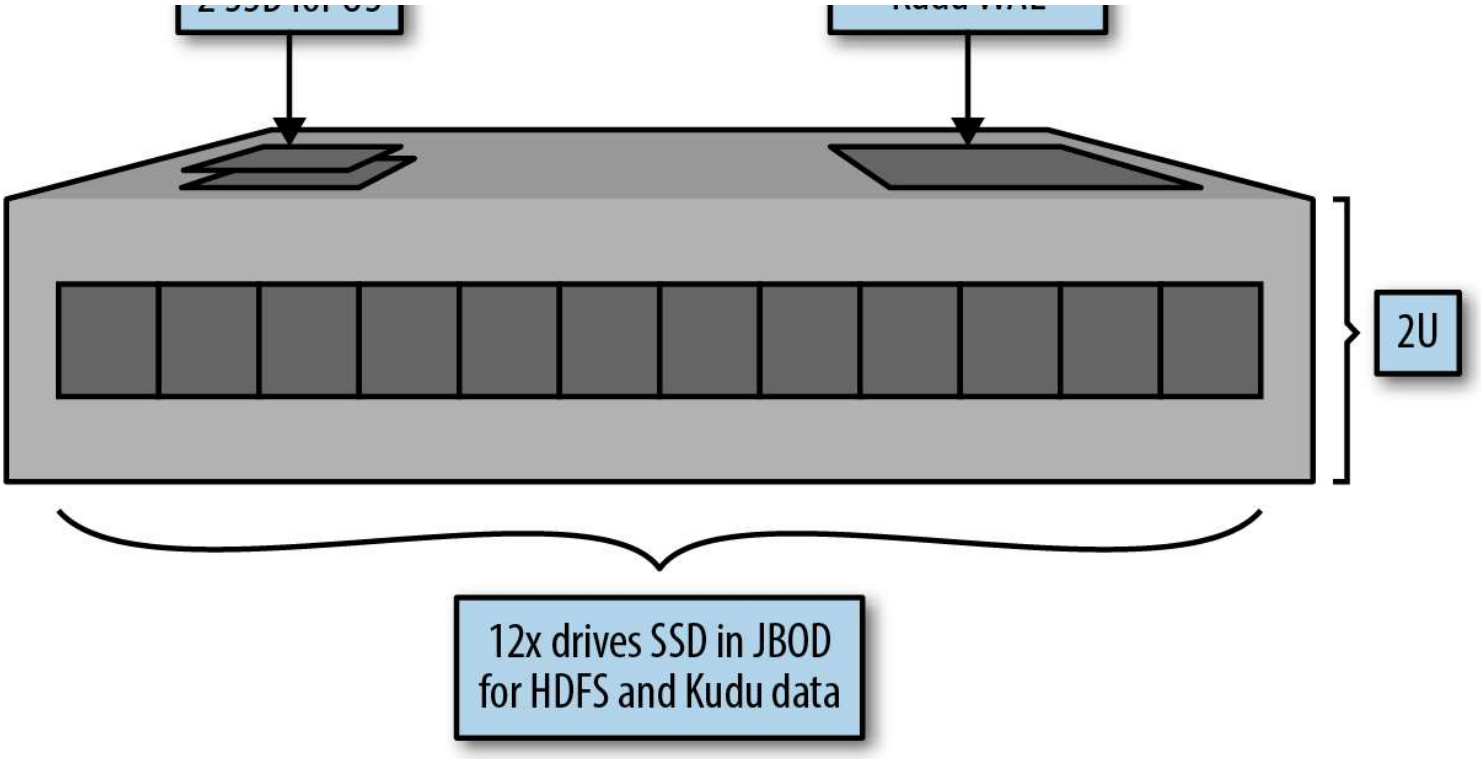


Figure 4-9. Adding NVMe PCIe flash storage for WAL

The new NVMe PCIe flash storage device is used for the WAL exclusively. Meanwhile, the Kudu data directories on the Kudu tablet servers would share the same filesystem mount points assigned to HDFS. Hence, consider the following filesystem mount points:

```
/data1
/data2
...
/data12
```

The path to HDFS would likely be something like:

```
/data1/dfs
...
/data12/dfs
```

We would essentially simply add Kudu directories a

This allows for clean separation between Kudu and HDFS in terms of directories managed by each service, while at the same time maximizing the utilization of the disk itself.

In Figure 4-10, we show how disk space might be occupied by the various services. To the OS, and even to the services themselves, they can detect simply how much space is left on the disk itself. Balancing and disk selection will automatically try to keep the disks approximately evenly utilized so that, at the end of the day, the disks will naturally be used up more and more evenly.

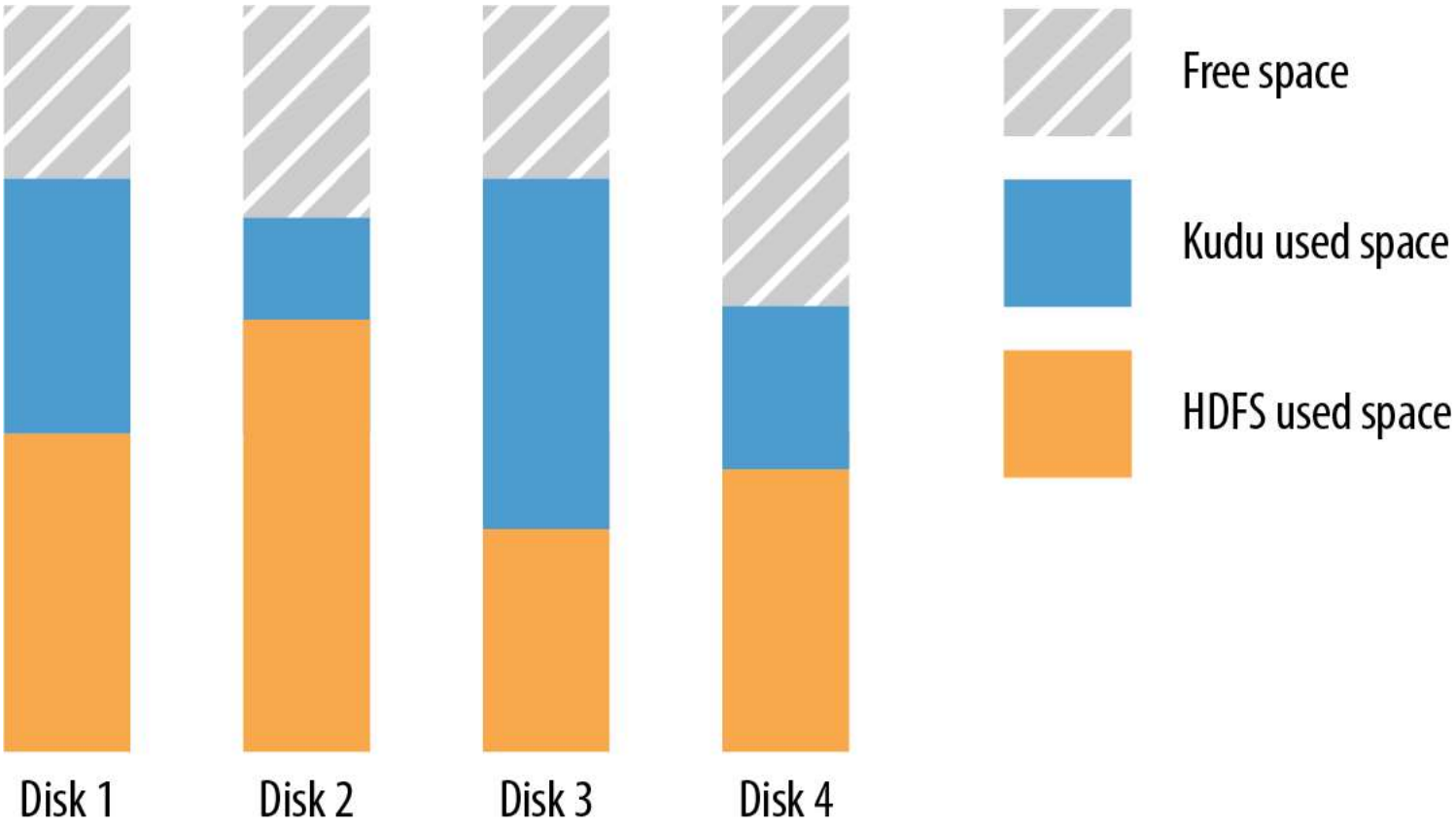


Figure 4-10. Recommended—use the same drives for both HDFS and Kudu

In general, we do not want to start disabling HDFS from using certain drives just so Kudu can have its own dedicated devices. If we did so, then the filesystem layout would look like this example:

```
/data1/dfs
...
/data6/dfs
```

This makes it very difficult to adjust requirements in the future, at which point we might end up having some disks that are extremely full for a given service, whereas the others might not be. Achieving a good balance is very difficult in this scenario. Figure 4-11 illustrates how that might end up looking.

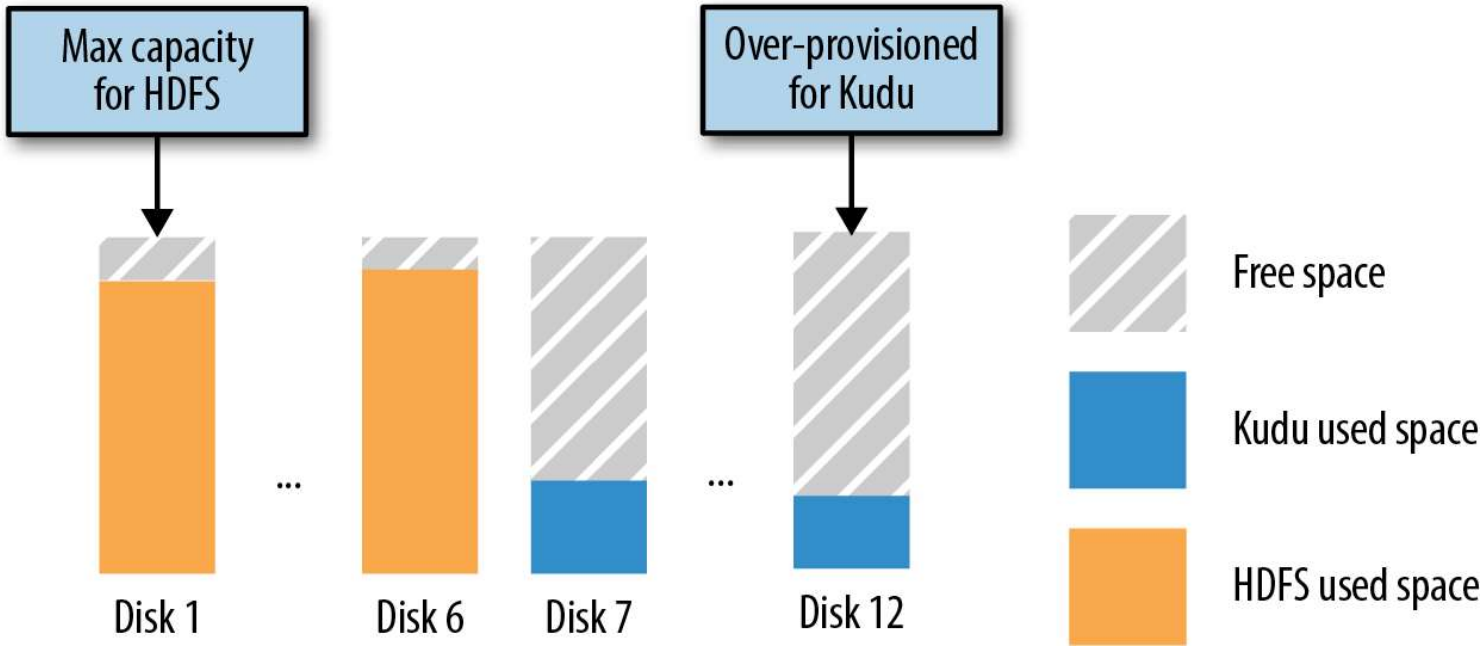


Figure 4-11. Discouraged—segregate HDFS and Kudu drives

If adding an NVMe–PCIe attached flash storage device for the WAL is not possible, it is at this point that we can consider putting the WAL on one of the existing data drives. However, in this case, we want to make sure that we use that drive exclusively for the WAL in Kudu; hence, it would be recommended to first remove that device from being used by HDFS. The transition would look something like Figure 4-12.

NOTE

Remember that disks in Hadoop are used not only by HDFS, but are also used as scratch directories for services such as YARN and Impala. If this is the case in your environment, you want to remove the usage of this disk drive from all existing services, so that the disk is fresh and dedicated to the Kudu WAL on the server.

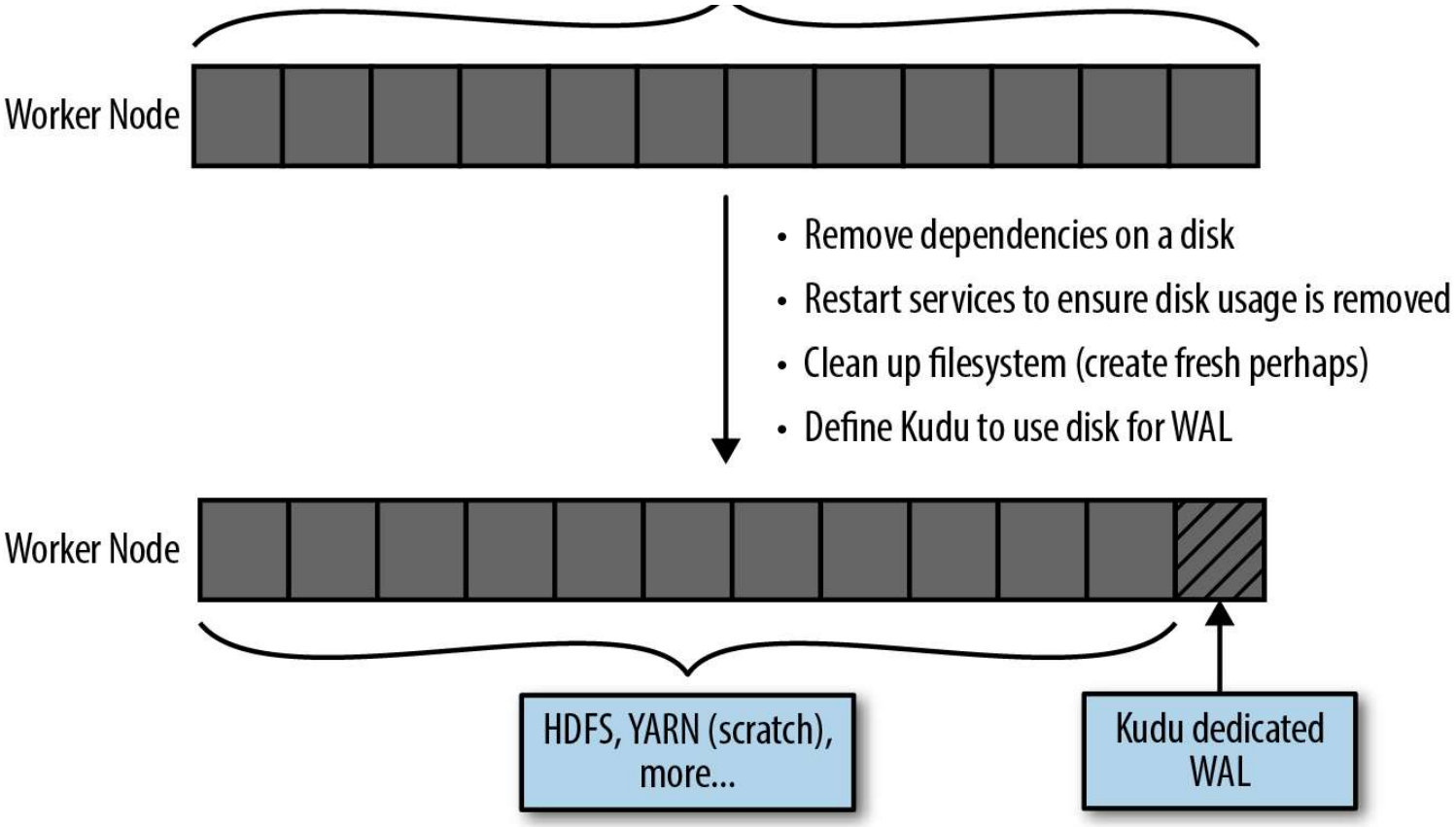


Figure 4-12. Transition worker node to also be tablet server

Of course, this retrofit is a little more painful when it is a DataNode and only 12 disks exist on the server. You lose capacity on every worker node, and typically, the WAL should not require the full capacity of the entire disk. In environments with 24 disks per worker node, it is likely much easier to give up a disk because they are typically smaller-capacity drives.

> **NOTE**
>
> Often, people attempt to add Kudu to a subset of worker nodes in an existing environment. Although technically doable, remember the caveats with this include a possible lopsided configuration between the worker nodes (some will have the 23 drives for HDFS/Kudu data + 1 for Kudu WAL, others will have 24 drives for HDFS), as well as encouraging more remote reads from Impala and Spark when processing from Kudu. Hence, this is generally discouraged.

Remember that a lot of the recommendations listed m
production clusters. In development environments, try the

Web UI of Tablet and Master Servers

Visually being able to quickly see how your Kudu deployment is configured, what tables exist, and where tablets reside is crucial to being an effective administrator.

Kudu ships with a web UI, which is the first place to assess these points.

Master Server UI and Tablet Server UI

The master server UI and tablet server Web UI have a common look and feel. Some of the commonalities include the capability to view the following:

Logs

: Last snippet of the log shown in the UI, and it shows a clear path to where you can find the logs on the server.

Memory

: Segregated into detailed memory breakdowns as well as summary totals to get a good understanding of where memory is being used.

Metrics

: An API endpoint intended to provide many metrics in JSON format, easily consumable by JSON parsers, to get at the metrics you like.

Remote procedure calls (RPCs)

: Listing of actively running and sampled remote procedure calls in JSON format.

Threads

: A view into threads, the amount of C… …ve I/O-wait measurements. Threads are categorized into thread groups, which

Flags

All of the flags defined at server startup time. This is a good way to validate whether your changes are taking effect as well as seeing the default settings at a glance.

Of course, master and tablet servers have different roles. Hence these UI elements are specific to the type of server you're analyzing.

Master Server UI

The master server UI has the following unique properties:

Masters

See the currently elected leader as well as the list of masters in your environment.

Tables

A list of all the tables.

Tablet servers

List of all the registered Tablet servers, their universal identifier, or UUID, along with RPC and HTTP address information and more.

Tablet Server UI

The tablet server UI has the following unique properties:

Dashboards

Provides a view into currently runni ntenanc operations being performed on the ta

The Kudu Command-Line Interface

Visuals are important for administrators to get a quick and easy overview of their environment. However, administrators often find themselves interested in a common set of detailed information and metrics as they zero in on what is most important to them. Thus, it makes a lot of sense for administrators to become familiar with the command-line interface (CLI) provided by Kudu. We summarize here some of the key operations that you can perform using the CLI in order to get to know your cluster well, understand the state that it is in, and make the right decisions when it comes to maintenance and other operations.

The CLI is accessed starting with the `kudu` executable followed by a command that determines what group of operations you'd like to perform. Commands are broken down into operations on the *cluster*, the *filesystem*, *local* and *remote* replicas, *metadata file* operations, *master and tablet server* operations, and table and WAL operations, while along the way checking on the health and integrity of the data itself.

Let's go through each group to see a few examples of what information you can obtain and monitor. We leave it to you to dig into the details of the commands in the Kudu documentation given that they are bound to change and evolve over time.

Cluster

At the cluster level, the only real command at your disposal is to perform a health check:

```
kudu cluster ksck <comma-separate-master-server-addresses>
```

The `comma-separate-master-server-addresses` might look as follows:

```
master-server1,master-server2
```

ships with, you typically should not need to specify the port number for these types of commands. If you change the port the master servers operate on, you will need to specify the port numbers in these commands.

Information returns the health state of all the tables and gathers information from all the tablet servers the masters know about. In particular, it provides information such as the following:

- Cluster health
- Data integrity
- Under-replicated tablets
- Unreachable tablet servers
- Tablets with no leader

You might want to perform checksums on your table data to ensure that the data written to disk is as expected. You can do so by using the following simple command:

```
kudu cluster ksck ip-172-31-48-12,ip-172-31-59-149  -checksum_scan
-tables=python-example
```

Performing checksum on your table is good practice especially after performing maintenance work or if there was an unexpected system outage in your data center. It helps to validate that your tables and the data within them did not become corrupted. More options are available to limit scans to specific tablet servers, improve concurrency of the checksums, and more.

Snapshot checksums, which are on by default, take [illegible] you have run this command. This is particularly hel[illegible]

This tool provides Kudu filesystem checks, formatting, and more. Kudu's notion of the "filesystem" is really a virtual concept that sits on top of your operating system's filesystem (such as `ext4`, `xfs`, etc) defined on the disks themselves. Exploring, formatting and checking the filesystem can be useful to have a better understanding of Kudu itself.

> **NOTE**
>
> The file *.gflagfile* contains the Kudu service startup options, which can sometimes be helpful to dig into how the Kudu processes were actually started. This is especially true if you set certain parameters but they do not seem to actually have any effect. In this file you can see whether your parameter was actually set as a startup flag.

check

Checking the filesystem for errors needs to be run as root. You must supply the *WAL* directory and then can provide a comma-separated list of data directories. An easy way to check what those directories are is to check the *.gflagfile* in use for your server.

You can find the *.gflagfile* by running the `ps` command:

```
ps -ef | grep kudu

# See output such as the following
kudu      10148      1  0 Aug15 ?        00:04:32
/usr/lib/kudu/sbin/kudu-tserver
--server_dump_info_path=/var/run/kudu/kudu-tserver-kudu.json
--flagfile=/etc/kudu/conf/tserver.gflagfile
```

Notice the `flagfile` option:

```
--flagfile=/etc/kudu/conf/tserver.gflagf
```


Now `grep` for the `fs` parameters in that file:

```
--fs_wal_dir=/var/lib/kudu/tserver
--fs_data_dirs=/var/lib/kudu/tserver
```

What we see here is that these directories are the same for the WAL and data directories. Kudu comes with filesystem "check" tools, which will provide a full block manager report that shows whether the Kudu filesystem is healthy. Things like missing blocks or orphaned blocks are warning signs that maybe something is not quite right with this filesystem.

When we do the `fs check` on the WAL directory, it (in this case) automatically finds the data directory as well as performs the checking for you:

```
sudo kudu fs check -fs_wal_dir=/var/lib/kudu/tserver/wals
```

The preceding output will then look something like this:

```
uuid: "e60fc0618b824f6a994748c053f9f4c2"
format_stamp: "Formatted at 2017-08-16 04:36:00 on ip-172-31-59-149.ec2.in
Block manager report
--------------------
1 data directories: /var/lib/kudu/tserver/data
Total live blocks: 0
Total live bytes: 0
Total live bytes (after alignment): 0
Total number of LBM containers: 0 (0 full)
Total missing blocks: 0
Total orphaned blocks: 0 (0 repaired)
Total orphaned block bytes: 0 (0 repaired)
Total full LBM containers with extra space: 0 (0 repaired)
Total full LBM container extra space in bytes: 0 (0 repaired)
Total incomplete LBM containers: 0 (0 repaired)
Total LBM partial records: 0 (0 repaired)
```

Now let's run the check on both the *WAL* and the *D*


```
sudo kudu fs check -fs_wal_dir=/var/lib/kudu/tserver
```

format

This prepares (formats) a brand-new Kudu filesystem.

Remember that this formatting is for the Kudu filesystem, not the operating system's filesystem. Kudu's filesystem is a set of user files and directories sitting on top of the OS filesystem, which means that you already need to have a directory or mount point prepared, with a supporting OS filesystem such as `ext3`. In our case, the `/` mount point is already on a basic `xfs`-formatted filesystem.

Let's see how our `/` filesystem is mounted currently:

```
$ mount | grep -E '^/dev'
/dev/xvda2 on / type xfs (rw,relatime,seclabel,attr2,inode64,noquota)
```

Here we see that the `/dev/xvda2` device mounted on `/` has been formatted as the `xfs` filesystem. We can also check the `/etc/fstab` file for the set of defined mount points.

For our purposes, we next prepare directories for our new Kudu *WAL* and *Data* directories, which we will then format using the Kudu format option:

```
# Create the WAL directory and assign ownership to kudu user
$ sudo mkdir /kudu-wal
$ sudo chown kudu:kudu /kudu-wal

# Create 8 directories for Data
$ for i in `seq 1 8`; do sudo mkdir /kudu-data$i;sudo chown kudu:kudu
/kudu-data$i; done
```

In this simple example, there's no real benefit to creating eight directories that are mounted on the same underlying disk. This is more relevant [illegible] tories, including the *WAL* directory, are written on a mount p[illegible]

Let's format these directories now for Kudu use:

The output you'll see is something along these lines:

```
I0821 22:33:10.068599  3199 env_posix.cc:1455] Raising process file limit
  1024 to 4096
I0821 22:33:10.068696  3199 file_cache.cc:463] Constructed file cache lbm
  capacity 1638
I0821 22:33:10.078719  3199 fs_manager.cc:377] Generated new instance meta
  in path /kudu-data1/instance:
uuid: "a7bc320ed46b47719da6c3b0073c74cc"
format_stamp: "Formatted at 2017-08-22 02:33:10 on ip-172-31-48-12.ec2.int
I0821 22:33:10.083366  3199 fs_manager.cc:377] Generated new instance meta
  path /kudu-data2/instance:
uuid: "a7bc320ed46b47719da6c3b0073c74cc"
...
I0821 22:33:10.141870  3199 fs_manager.cc:377] Generated new instance meta
  path /kudu-wal/instance:
uuid: "a7bc320ed46b47719da6c3b0073c74cc"
format_stamp: "Formatted at 2017-08-22 02:33:10 on ip-172-31-48-12.ec2.int
```

Notice that this single filesystem is registered according to a given `uuid`. In this case, the `uuid` is `a7bc320ed46b47719da6c3b0073c74cc`. You also can specify this `uuid` when creating the formatted data directory.

dump

The dump command enables you to dump various portions of the filesystem.

We begin by dumping the `uuid` of the filesystem, which we get by specifying the *WAL* and *Data* directories:

```
sudo kudu fs dump uuid -fs_wal_dir=/v
```


This will dump a little information, where the relev

```
_
3505a1efac6b4bdc93a5129bd7cf624e
```

Next, we want to get more information about this filesystem. We can dump it at the block level and see a lot more information:

```
$ sudo kudu fs dump tree -fs_wal_dir=/var/lib/kudu/tserver
```

This gives us information such as the following:

```
1 data directories: /var/lib/kudu/tserver/data
Total live blocks: 6
Total live bytes: 8947
Total live bytes (after alignment): 32768
Total number of LBM containers: 5 (0 full)
...
uuid: "3505a1efac6b4bdc93a5129bd7cf624e"
format_stamp: "Formatted at 2017-08-15 17:08:52 on ip-172-31-48-12.ec2.int
File-System Root: /var/lib/kudu/tserver
|-instance
|-wals/
|----4322392e8d3b49538a959be9d37d6dc0/
|-------wal-000000001
|-------index.000000000
|----15346a340a0841798232f2a3a991c35d/
|-------wal-000000001
|-------index.000000000
|----7eea0cba0b854d28bf9c4c7377633373/
|-------wal-000000001
|-------index.000000000
|-tablet-meta/
|----4322392e8d3b49538a959be9d37d6dc0
|----15346a340a0841798232f2a3a991c35d
|----7eea0cba0b854d28bf9c4c7377633373
|-consensus-meta/
|----4322392e8d3b49538a959be9d37d6dc0
|----15346a340a0841798232f2a3a991c35d
|----7eea0cba0b854d28bf9c4c7377633373
|-data/
```

```
|    eb44ed9bbcce479c89908fbe939ccf49.metadata
|----eb44ed9bbcce479c89908fbe939ccf49.data
|----9e4d11c4b6ac4e1ea140364d171e5e7b.metadata
|----9e4d11c4b6ac4e1ea140364d171e5e7b.data
|----80a281a920fc4c10b79ea7d2b87b8ded.metadata
|----80a281a920fc4c10b79ea7d2b87b8ded.data
|----2f5bc79e814f4a6a98b12a79d5994e99.metadata
|----2f5bc79e814f4a6a98b12a79d5994e99.data
```

You can find the `block_id` by using the following dump commands on the local or remote replicas. Columns are broken down into blocks which each have their own block ID. In the next section on dumping Tablet information, you can find how to get the block ID, then you can dump the contents in human-readable format with the following command:

```
$ sudo kudu fs dump cfile 0000000000000007 -fs_wal_dir=/var/lib/kudu/tserv
Header:

Footer:
data_type: INT64
encoding: BIT_SHUFFLE
num_values: 1
posidx_info {
  root_block {
    offset: 51
    size: 19
  }
}
validx_info {
  root_block {
    offset: 70
    size: 16
  }
}
compression: NO_COMPRESSION
metadata {
  key: "min_key"
  value: "\200\000\000\000\000\000\000\0
}
metadata {
```

```
is_type_nullable: false
incompatible_features: 0

1
```

As an administrator, this gives visibility into the actual block showing various details about data types, encoding strategies, compression settings, min/max keys and more.

Tablet Replica

You can perform tablet replica operations either locally or remotely by using the `local_replica` or `remote_replica` operations, respectively. These operations are used to take a slightly higher up view from the blocks themselves that we saw at the end of the last section. A `remote_replica` command is useful here because you don't have to log in to each and every machine just to run the commands.

Let's begin by listing the replicas we have on our server:

```
sudo kudu local_replica list -fs_wal_dir=/var/lib/kudu/tserver
...
4322392e8d3b49538a959be9d37d6dc0
15346a340a0841798232f2a3a991c35d
7eea0cba0b854d28bf9c4c7377633373
```

We can inspect one of these local replicas by dumping information about it. Let's begin with the `block_ids` themselves:

```
sudo kudu local_replica dump block_ids 15346a340a0841798232f2a3a991c35d
-fs_wal_dir=/var/lib/kudu/tserver
Rowset 0
Column block for column ID 0 (key[int64 NOT NULL]): 0000000000000007
Column block for column ID 1 (ts_val[unixtime micros NOT NULL]): 000000000
```


We can further dump information about the metadata of the table replica.

```
Table name: python-example Table id: d8f1c3b488c1433294c3daf0af4038c7
Schema (version=0): Schema [
        0:key[int64 NOT NULL],
        1:ts_val[unixtime_micros NOT NULL]
]
Superblock:
table_id: "d8f1c3b488c1433294c3daf0af4038c7"
tablet_id: "15346a340a0841798232f2a3a991c35d"
last_durable_mrs_id: 0

rowsets {
  id: 0
  last_durable_dms_id: -1
  columns {
    block {
      id: 7
    }
    column_id: 0
  }
  columns {
    block {
      id: 8
    }
    column_id: 1
  }
  bloom_block {
    id: 9
  }
}
table_name: "python-example"
schema {
  columns {
    id: 0
    name: "key"
    type: INT64
    is_key: true
    is_nullable: false
    encoding: AUTO_ENCODING
    compression: DEFAULT_COMPRESSION
    cfile_block_size: 0
  }
  columns {
    id: 1
```

```
      is_nullable: false
      encoding: AUTO_ENCODING
      compression: LZ4
      cfile_block_size: 0
    }
  }
  schema_version: 0
  tablet_data_state: TABLET_DATA_READY
  orphaned_blocks {
    id: 10
  }
  partition {
    hash_buckets: 1
    partition_key_start: "\000\000\000\001"
    partition_key_end: ""
  }
  partition_schema {
    hash_bucket_schemas {
      columns {
        id: 0
      }
      num_buckets: 2
      seed: 0
    }
    range_schema {
      columns {
        id: 0
      }
    }
  }
```

In this block of replica metadata output, we have preamble content describing the table itself, such as its name and schema. Next, *rowset* information is provided on a per-column level followed by a detailed description of the table schema, per column. The final portion of the dump shows us the state of this tablet, and provides details about the partitioning schema that this replica uses.

Next, let's go a bit deeper, looking into the rowset i

```
-------------------------------------------------------------------------
  RowSet metadata: id: 0
last_durable_dms_id: -1
columns {
  block {
    id: 7
  }
  column_id: 0
}
columns {
  block {
    id: 8
  }
  column_id: 1
}
bloom_block {
  id: 9
}

  Dumping column block 0000000000000007 for column id 0( key[int64 NOT NULL
  -------------------------------------------------------------------------
  CFile Header:

  Dumping column block 0000000000000008 for column id 1( ts_val[unixtime_m
                                                          NOT NULL]):
  -------------------------------------------------------------------------
  CFile Header:
```

We get similar information as in the replica dump, beginning with seeing how the rowset looks per column, and then we gain detailed insight for each column block that exists for the rowset dump.

Finally, we dump out the WAL information for this replica and are privy to seeing a sequence number in the WAL, followed by schema information, compression details, a set of operations that occurred, and information about the minimum and maximum replica indices:


```
$ sudo kudu local_replica dump wals  153                    3
-fs_wal_dir=/var/lib/kudu/tserver
```

```
schema {
  columns {
    id: 0
    name: "key"
    type: INT64
    is_key: true
    is_nullable: false
    encoding: AUTO_ENCODING
    compression: DEFAULT_COMPRESSION
    cfile_block_size: 0
  }
  columns {
    id: 1
    name: "ts_val"
    type: UNIXTIME_MICROS
    is_key: false
    is_nullable: false
    encoding: AUTO_ENCODING
    compression: LZ4
    cfile_block_size: 0
  }
}
schema_version: 0
compression_codec: LZ4
1.1@6155710131387969536 REPLICATE NO_OP
        id { term: 1 index: 1 } timestamp: 6155710131387969536 op_type:
          NO_OP noop_request { }
COMMIT 1.1
        op_type: NO_OP commited_op_id { term: 1 index: 1 }
1.2@6155710131454771200 REPLICATE WRITE_OP
        Tablet: 15346a340a0841798232f2a3a991c35d
        RequestId: client_id: "1ffdc96b901545468e37569a262d3bd1" seq_no:
          1 first_incomplete_seq_no: 0 attempt_no: 0
        Consistency: CLIENT_PROPAGATED
        op 0: INSERT (int64 key=1, unixtime_micros
                      ts_val=2017-08-16T04:48:38.803992Z)
        op 1: MUTATE (int64 key=1) SET ts_val=2017-01-01T00:00:00.000000Z
COMMIT 1.2
        op_type: WRITE_OP commited_op id { term: 1 index: 2 } result { ops
          { skip_on_replay: true mutated
          { skip_on_replay: true mutated
2.3@6157789003386167296 REPLICATE NO_OP
        id { term: 2 index: 3 } timestamp: 6157789003386167296 op_type:
```

```
num_entries: 6
min_replicate_index: 1
max_replicate_index: 3
```

We can also look at the `remote_replica` command, and what we'll do is copy one of those replicas over to this node.

First, we take a look at the list of tablet servers we have so that we can move a replica from one of those servers to our local server:

```
$ kudu tserver list ip-172-31-48-12
                uuid                 |           rpc-addresses
-------------------------------------+-----------------------------------
 bb50e454938b4cce8a6d0df3a252cd44 | ip-172-31-54-170.ec2.internal:7050
 060a83dd48e9422ea996a8712b21cae4 | ip-172-31-56-0.ec2.internal:7050
 3505a1efac6b4bdc93a5129bd7cf624e | ip-172-31-48-12.ec2.internal:7050
 e60fc0618b824f6a994748c053f9f4c2 | ip-172-31-59-149.ec2.internal:7050
```

Now let's do a remote replica listing (Figure 4-13). This is different from a `local_replica` listing because we see not only the list of tablet IDs, but we see the state, table name, partition, estimated on-disk size, and schema:

```
$ sudo kudu remote_replica list ip-172-31-56-0.ec2.internal:7050
Tablet id: 7eea0cba0b854d28bf9c4c7377633373
State: RUNNING
Table name: python-example-repl2
Partition: HASH (key) PARTITION 3, RANGE (key) PARTITION UNBOUNDED
Estimated on disk size: 310B
Schema: Schema [
        0:key[int64 NOT NULL],
        1:ts_val[unixtime_micros NOT NULL]
]
Tablet id: ca921612aeac4516886f36bab0415
State: RUNNING
Table name: python-example-repl2
Partition: HASH (key) PARTITION 1, RANGE (key) PARTITION UNBOUNDED
```

```
        1:ts_val[unixtime_micros NOT NULL]
]
Tablet id: 4e2dc9f0cbde4597a7dca2fd1a05f995
State: RUNNING
Table name: python-example-repl2
Partition: HASH (key) PARTITION 2, RANGE (key) PARTITION UNBOUNDED
Estimated on disk size: 0B
Schema: Schema [
        0:key[int64 NOT NULL],
        1:ts_val[unixtime_micros NOT NULL]
]
Tablet id: 1807d376c9a044daac9b574e5243145b
State: RUNNING
Table name: python-example-repl2
Partition: HASH (key) PARTITION 0, RANGE (key) PARTITION UNBOUNDED
Estimated on disk size: 310B
Schema: Schema [
        0:key[int64 NOT NULL],
        1:ts_val[unixtime_micros NOT NULL]
]
```

TIP

For the purpose of simplicity in the following figures, we make note of the last three alphanumeric characters in the Tablet IDs listed in the previous code example so it is easier to identify which tablet we are describing.

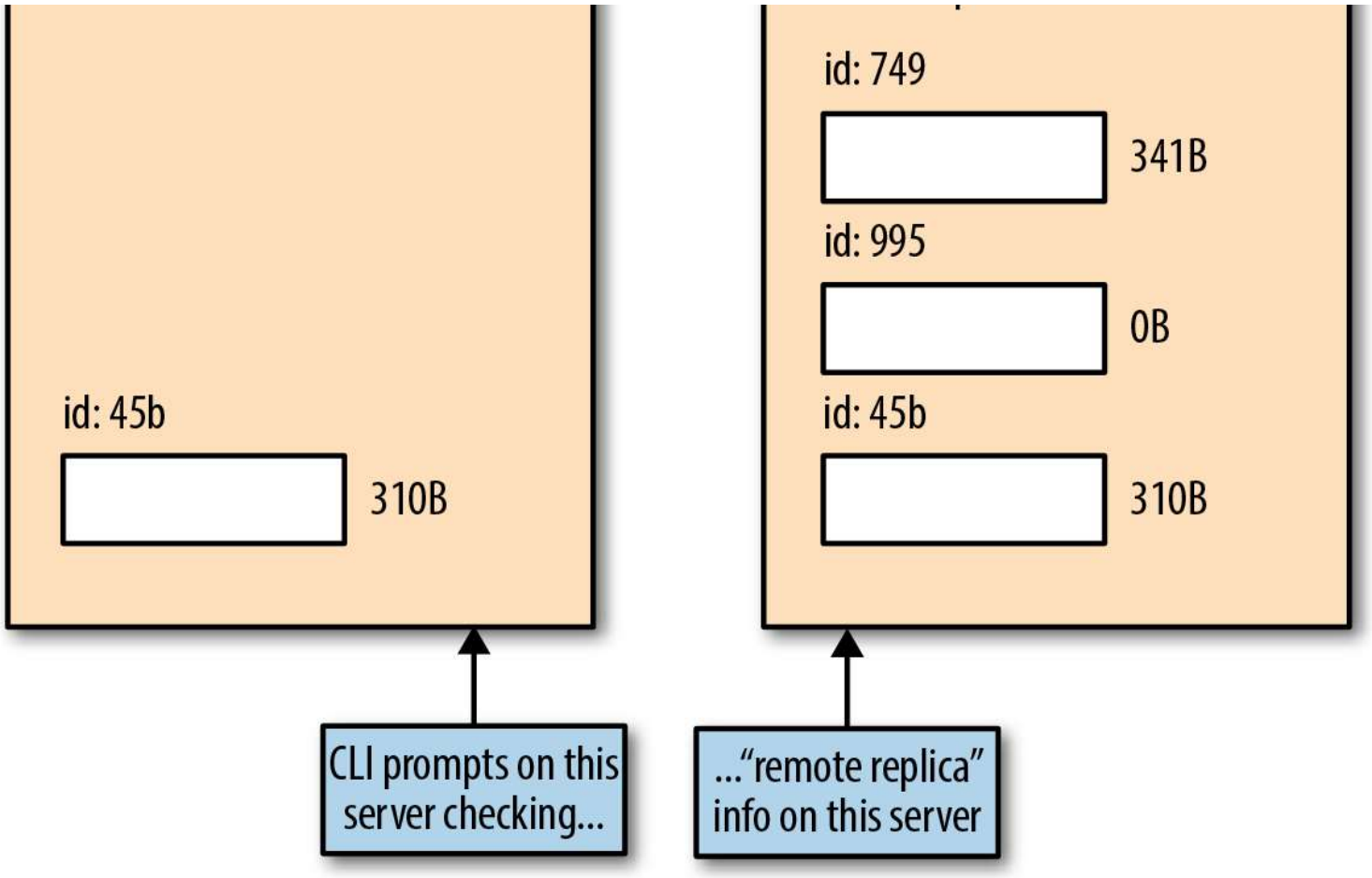


Figure 4-13. Remote replica listing

Copy a remote replica to a local server

To copy a replica from a remote server, you need to stop the local server on which you're running. Then, the copy command is in fact making a connection to the remote, up and running server, and making a copy of the replica locally.

After you're logged into the local server, stop the server by using the following command:

```
sudo systemctl stop kudu-tserver
```

Suppose that we want to copy over Tablet ID `1807d376c9a044daac9b574e5243145b` that is on remote_replica ip-172-31-56-0. We already have that on our local replica. Let's see how this works.

```
I0821 23:43:38.586378  3078 tablet_copy_client.cc:166] T
1807d376c9a044daac9b574e5243145b P 3505a1efac6b4bdc93a5129bd7cf624e:
Tablet Copy client: Beginning tablet copy session from remote peer at addr
  ip-172-31-56-0.ec2.internal:7050
Already present: Tablet already exists: 1807d376c9a044daac9b574e5243145b
```

NOTE

You must run the `copy_from_remote` command of `local_replica` must be run as the `kudu` user itself to have appropriate permissions to perform the task. You can't do this as `root` or as a `sudo` user.

The error message is clear that if you have a replica of a given tablet on the remote server but also already present on the local server, it won't actually allow you to copy it—you already have it (Figure 4-14).

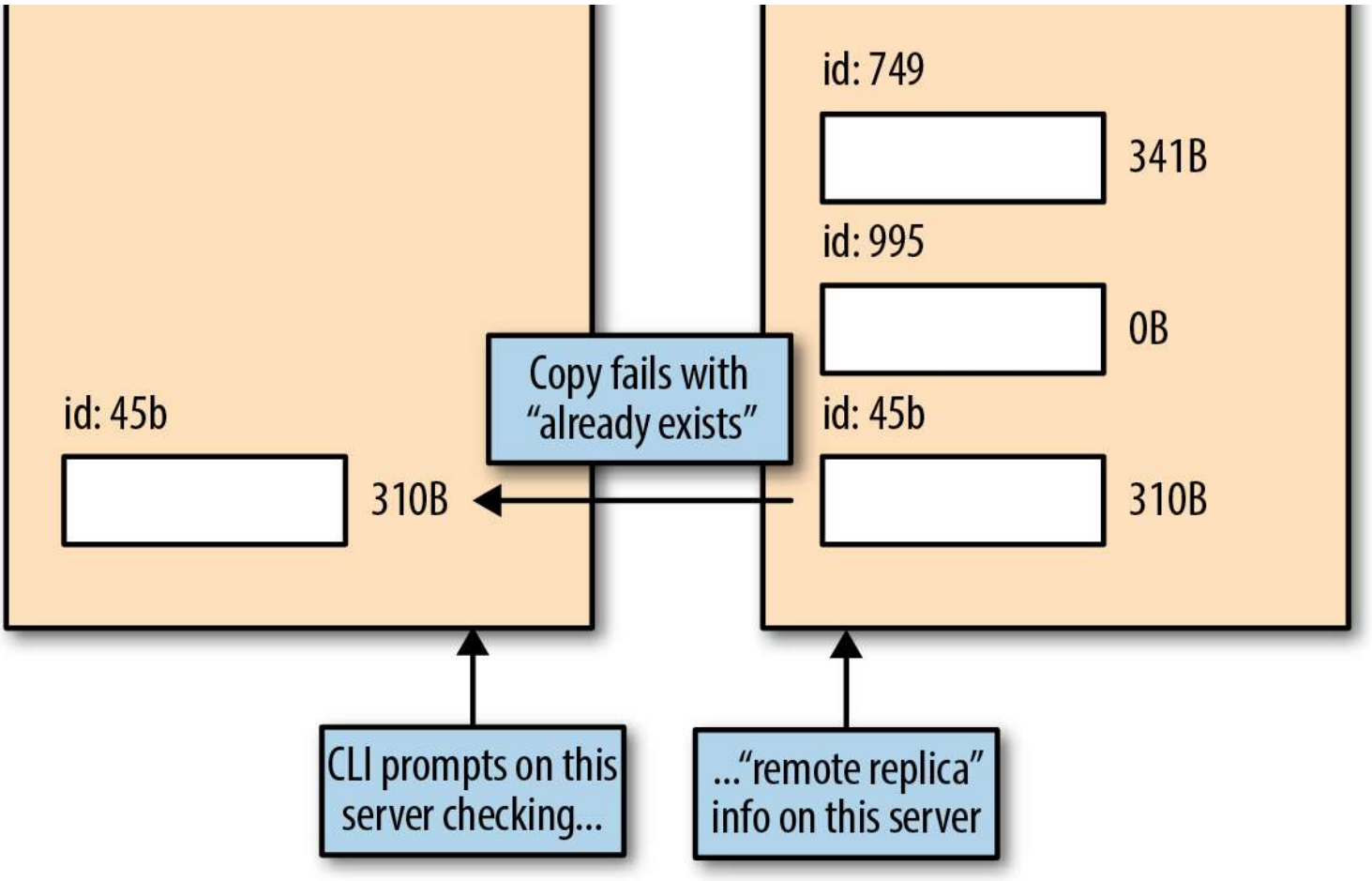


Figure 4-14. Failed copy with an alert that the tablet already exists

Let's now pick a replica that we do not have on our local server. In our case, `PARTITION 1` is found only on the remote server, not locally. It is represented by ID `ca921612aeac4516886f36bab0415749`. Let's try to copy that:

```
$ kudu local_replica copy_from_remote ca921612aeac4516886f36bab0415749
  ip-172-31-56-0.ec2.internal:7050 -fs_wal_dir=/var/lib/kudu/tserver
I0821 23:55:19.530689  3971 tablet_copy_client.cc:166] T
  ca921612aeac4516886f36bab0415749 P 3505a1efac6b4bdc93a5129bd7cf624e:
  Tablet Copy client: Beginning tablet copy session from remote peer at
  address ip-172-31-56-0.ec2.internal:7050
I0821 23:55:19.542532  3971 tablet_copy_client.cc:422] T
  ca921612aeac4516886f36bab0415749 P 3505a1efac6b4bdc93a5129bd7cf624e:
  Tablet Copy client: Starting download of 3 data blocks...
I0821 23:55:19.562101  3971 tablet_copy_client.cc:385] T
  ca921612aeac4516886f36bab0415749 P 350                    624e:
  Tablet Copy client: Starting download
I0821 23:55:19.566256  3971 tablet_copy_
```

We've now copied the data successfully. At this point, we have four replicas of this dataset as it has been active on three replicas already.

However, although the tablet has been copied, it doesn't immediately take part in the Raft consensus. Consequently, it doesn't actually have any role yet; it can be neither a follower nor a leader of this local replica.

Let's take a look at our table again and the specific set of tablets we have. The second column next to the `T` gives us the tablet ID, whereas the entries in the row give us the replica UUIDs:

> **NOTE**
>
> In our shell, we've exported the variable `KMASTER` with the list of Kudu master servers.

```
[ec2-user@ip-172-31-48-12 ~]$ kudu table list $KMASTER -list_tablets
python-example-repl2
T 1807d376c9a044daac9b574e5243145b
  P    060a83dd48e9422ea996a8712b21cae4(ip-172-31-56-0.ec2.internal:7050)
  P(L) e60fc0618b824f6a994748c053f9f4c2(ip-172-31-59-149.ec2.internal:7050
  P    bb50e454938b4cce8a6d0df3a252cd44(ip-172-31-54-170.ec2.internal:7050
T ca921612aeac4516886f36bab0415749
  P(L) e60fc0618b824f6a994748c053f9f4c2(ip-172-31-59-149.ec2.internal:7050
  P    060a83dd48e9422ea996a8712b21cae4(ip-172-31-56-0.ec2.internal:7050)
  P    bb50e454938b4cce8a6d0df3a252cd44(ip-172-31-54-170.ec2.internal:7050
T 4e2dc9f0cbde4597a7dca2fd1a05f995
  P(L) e60fc0618b824f6a994748c053f9f4c2(ip-172-31-59-149.ec2.internal:7050
  P    060a83dd48e9422ea996a8712b21cae4(ip-172-31-56-0.ec2.internal:7050)
  P    bb50e454938b4cce8a6d0df3a252cd44(ip-172-31-54-170.ec2.internal:7050
T 7eea0cba0b854d28bf9c4c7377633373
  P(L) e60fc0618b824f6a994748c053f9f4c2(ip-172-31-59-149.ec2.internal:7050
  P    060a83dd48e9422ea996a8712b21cae4(ip-172-31-56-0.ec2.internal:7050)
  P    bb50e454938b4cce8a6d0df3a252cd44(ip-172-31-54-170.ec2.internal:7050
```

```
#
$ kudu tablet change_config remove_replica $KMASTER
  ca921612aeac4516886f36bab0415749 060a83dd48e9422ea996a8712b21cae4

# Now add the replica on the 12 server
kudu tablet change_config change_replica_type $KMASTER
ca921612aeac4516886f36bab0415749 060a83dd48e9422ea996a8712b21cae4 NON-VOTE

kudu tablet change_config add_replica $KMASTER ca921612aeac4516886f36bab04
```

Figure 4-15 depicts what we just performed. We identified the tablet replica we wanted to copy locally. By stopping the local tablet server first, we copied the tablet, disabled the old replica, activated the current one, and then we restart the tablet server.

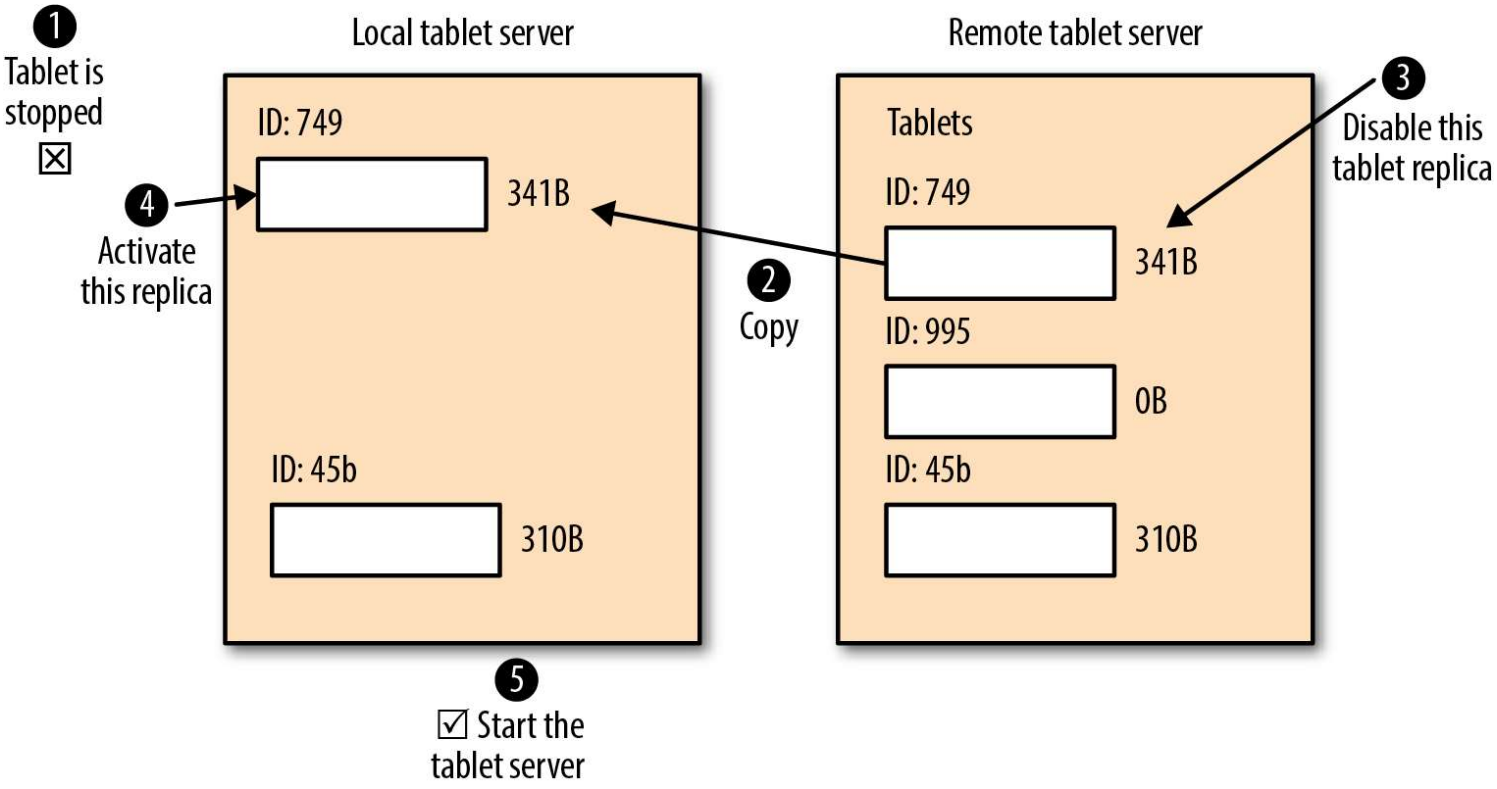


Figure 4-15. Copying a remote replica locally

Deleting a replica

Using the `local_replica delete` option, you c a tablet server. To do so, you first need to stop the tablet server and then carry out the `delete`

```
# Stop the tablet server
sudo service kudu-tserver stop

# List the replicas on your tablet server
sudo kudu local_replica list --fs_wal_dir=/var/lib/kudu/tserver
4322392e8d3b49538a959be9d37d6dc0
15346a340a0841798232f2a3a991c35d
ca921612aeac4516886f36bab0415749

# Delete the one you prefer
sudo kudu local_replica delete 4322392e8d3b49538a959be9d37d6dc0
  --fs_wal_dir=/var/lib/kudu/tserver
uuid: "3505a1efac6b4bdc93a5129bd7cf624e"
format_stamp: "Formatted at 2017-08-15 17:08:52 on ip-172-31-48-12.ec2.int
I0915 17:26:11.767314  2780 ts_tablet_manager.cc:1063]
  T 4322392e8d3b49538a959be9d37d6dc0 P 3505a1efac6b4bdc93a5129bd7cf624e:
  Deleting tablet data with delete state TABLET_DATA_TOMBSTONED
I0915 17:26:11.773490  2780 ts_tablet_manager.cc:1075]
  T 4322392e8d3b49538a959be9d37d6dc0 P 3505a1efac6b4bdc93a5129bd7cf624e:
  Tablet deleted. Last logged OpId: 7.8
I0915 17:26:11.773535  2780 log.cc:965]
  T 4322392e8d3b49538a959be9d37d6dc0
  P 3505a1efac6b4bdc93a5129bd7cf624e: Deleting WAL directory at
  /var/lib/kudu/tserver/wals/4322392e8d3b49538a959be9d37d6dc0
```

Notice that the tablet data is actually set to the `TABLET_DATA_TOMBSTONED` state. We can see from the log message that the WAL has been deleted completely. Data from your replica is likewise cleaned up; however, consensus and tablet metadata information remains by default. This is to ensure that Raft vote durability requirements remain intact.

To fully remove all the metadata as well (which is considered an unsafe operation because if it is not done properly then it may yield harmful side effects), you can add the `--clean_unsafe=true` option to the `delete` operation, and all the replica metadata information will be removed.

Consensus Metadata

You can also try metadata for consensus informatio

Let's first list the set of replicas on our host:

```
# List replicas on this tablet server
sudo kudu local_replica list --fs_wal_dir=/var/lib/kudu/tserver
15346a340a0841798232f2a3a991c35d
ca921612aeac4516886f36bab0415749
```

Now we can see the peer UUIDs in Raft's configuration by picking one of the replicas:

```
# Peer UUIDs for Raft configuration
sudo kudu local_replica cmeta print_replica_uuids
  ca921612aeac4516886f36bab0415749
  --fs_wal_dir=/var/lib/kudu/tserver
e60fc0618b824f6a994748c053f9f4c2 060a83dd48e9422ea996a8712b21cae4
  bb50e454938b4cce8a6d0df3a252cd44
```

Hence, we can see three UUIDs of peer replicas. This example is actually from a tombstoned replica. As a result, we can still see the Raft consensus information stored on this tablet server, and we would need to perform a delete with the `--clean_unsafe` operation in order to fully clean up this information.

You also can rewrite a tablet's Raft configuration by using the `rewrite_raft_config` operation on the `cmeta` data. We leave it to you to check the documentation and proceed with caution; this type of action should be performed only by advanced users in situations when the system might be down.

Similarly you can change Raft replication configurations on a remote tablet server. By passing in the remote tablet server, the tablet ID, and the set of peer UUIDs, you can overwrite the Raft consensus configuration. You can use this to recover from problematic scenarios such as a loss of a majority of replicas. But be aware that there is the risk of losing edits. See the `kudu remote_replica unsafe_change_config` operation for details.

Adding and Removing Tablet Serve

servers as required.

Adding Tablet Servers

Adding new tablet servers typically just means installing the necessary binaries for tablet servers and making configuration changes to inform those tablet servers with which master servers it needs to register.

Here is an example of installing binaries. Follow the Quick Install instructions from Chapter 3 to set up the Kudu repository. Then, install only the packages required for the tablet server:

```
sudo yum -y install kudu              # Base Kudu files (all nodes)
sudo yum -y install kudu-tserver      # Kudu tablet server (tablet nodes o
```

In our three-minute tutorial, we didn't bother to make any adjustments to configuration files, because everything was installed on the `localhost`. This time, we've installed only the tablet server, with no master server installed on this host. So let's set a few configuration parameters on this tablet server.

You can find the primary file that contains flags and options under */etc/kudu/conf/tserver.gflagfile*.

Open that file in your favorite editor and append the following line:

```
--tserver_master_addrs=ip-172-31-48-12.ec2.internal:7051
```

Now start the tablet server service itself:

```
sudo systemctl start kudu-tserver
```

```
sudo vi /var/log/kudu/kudu-tserver.INFO
```

You should see a similar command to the following:

```
sudo grep Register /var/log/kudu/kudu-tserver.INFO
I0816 00:41:36.482281 10669 heartbeater.cc:361] Registering TS with master
```

While at the top of the log file, you'll see the parameters you specified as flags for the service.

Removing a Tablet Server

Removing a tablet server basically involves decommissioning them than removing the packages associated with Kudu on that node.

To decommission a tablet server gracefully, it is safest to take the following steps:

1. Copy all the replicas from the tablet server to another live tablet server and ensure it is part of the Raft concensus
2. Remove all replicas from the given tablet server
3. Stop the tablet server
4. Remove tablet server binaries

To ensure that no new tablets are being created on the tablet server while performing the decommissioning operation, you will need to simply ensure that no new DDL is taking place at the time of the removal.

You can find the full list of options for configuring

thought. Security is typically thought about in the following three ways:

Authentication

: A guarantee that the person attempting access is who they say they are.

Authorization

: A mode of providing access to the given person, whether that be to a given technology or data.

Encryption

: Data flowing over the network and/or stored at rest is encrypted.

We go through some of these concepts using a simple analogy before we get into configuring Kudu with security.

A Simple Analogy

A simple example of authentication in everyday life is handing over your passport when passing through security at an airport. The border guard confirms that you are the passport holder by comparing the photo in the passport with you standing right in front of them. Passing this checkpoint means that you've been "authenticated," or in other words, they have validated that you are who you say you are.

In this airport analogy, as you progress to your gate for boarding, the agents at the gate validate that you indeed are allowed to board the plane. Your boarding pass allows you to board only the airplane for which you have purchased a ticket. This is a form of *authorization* because you are authorized, and authorized only to board that one plane, which is cross-checked by the gate agents.

Now, suppose that you had a letter you were taking [illegible] to be kept a secret so that only your mom can read it when you [illegible] contents of your letter while you were still at home, [illegible] on the way

In the world of data management, "boarding the plane" would be the same as access to a database or a table within a database, or a file within a filesystem, for example. Meanwhile, the border guard authenticating you performs the similar job of the industry standard big data strong authentication mechanism, Kerberos. Finally, the scrambled contents of your letter represent encryption in transit, and perhaps at rest, as well, as you leave the letter at your mom's house.

We can further think through this analogy by coming to the realization that authentication is something you, as a client, are responsible for in the sense that you needed to bring your passport with you. Further, you as a client also need to make a judgment call that indeed the border guard is truly a representative of the border patrol for the given country. We intuitively know this based on the uniform or badge they wear. We can then know the agency that issued the border guard's badge from prior knowledge.

Hence, from a "passenger" or "client-side" perspective, there are two pieces of information we have access to on our end:

- A passport proving who we are
- Prior knowledge of the authority that issued the badge and uniform to the border officer

From "airport/airline" or "server-side" perspective, we need to be configured to do the following:

- Agree to allow passport checks as the mode of validating who you are
- Have border guards who wear the right badge and uniform
- Have knowledge about which plane you a

On the client side, the client needs to do the following:

- Be Kerberos enabled and have the Kerberos Ticket Granting Ticket (TGT) when accessing Kudu (i.e., the passport)
- Have knowledge of the Certificate Authority (CA) for validating Transport Layer Security (TLS) certificates configured on the Kudu end (i.e., knowledge of the authority for the guard's badge)

On the server side, the server needs to do the following:

- Be enabled for Kerberos authentication (i.e., agree to a passport being the mode of validation and authentication)
- Have TLS certificates that were signed by an appropriate CA (i.e., border guards wearing the right badge and uniform)
- Have an authorization database of who has access to which table, which would typically be done by a service such as Sentry (i.e., knowledge about the plane you're allowed to board, which is stored in a database by the airline)

With this primer under our belts, we can go through the capabilities Kudu has today with regard to security and where it is headed in the near future.

Kudu Security Features

Kudu's support of security features is continually increasing, and at the time of this writing, it supports the following:

- Encryption over the wire, using TLS
- Data at rest encryption
- Kerberos authentication

- Log redaction
- Web UI security

Encryption over the wire

Encryption over the wire needs to be looked at from two different perspectives. The first is with *internal* communication between tablet and master servers within the cluster. Kudu has a built-in mechanism to build and issue internal X.509 certificates to all the servers in the cluster. Hence, TLS is the primary mechanism used in encrypting traffic over the wire, internal to the cluster.

These certificates serve a second purpose for intracluster communication. They in fact offer strong authentication using this mechanism so that each server within the cluster can trust that the service connecting to it, whether it be a tablet server or master server, is in fact who they say they are.

Normally, Kerberos is used for authentication, though in this particular case, using the certificate strategy for intracluster communication allows for less load on the Kerberos Key Distribution Center (KDC). This allows for the cluster to scale really well in terms of not having to go to the KDC for each service in the cluster for authentication purposes; rather, they rely on certificates to guarantee authentication for intracluster server communication.

Encryption between Kudu clients and servers is also enabled by using this very same mechanism. Hence, enabling encryption for over-the-wire communication and authentication between servers is as simple as setting the flags:

```
--rpc_authentication=required
--rpc_encryption=required
```

ing the master and tablet servers or as entries in the flagfile during startup. Kudu services on all nodes must be started with this set of flags to properly take effect.

These flags are set when the master or tablet servers are started. Here is an example specifying them directly in the command-line options, you can also specify them in the `gflagfile` mentioned by the `--flagfile` option:

```
kudu-master --rpc_authentication=required --rpc_encryption=required
--master_addresses=... --flagfile=gflagfile
```

Alternatively, if you're running Kudu in Cloudera, it is as simple as setting the `enable_security` flag in Cloudera Manager for the Kudu service, labeled as "Enable Secure Authentication and Encryption"; the preceding two parameters will be automatically set during startup.

We note here that `--rpc_encryption` has other valid options such as `disabled` and `optional`, though usage of those is not recommended. The `optional` flag will have Kudu attempt to use encryption, and if it fails will allow unencrypted traffic only from trusted subnets. We leave it to you to look more into these options from the Kudu documentation, as in general, we discourage this practice.

Data-at-rest encryption

Data at rest encryption is not possible at this time by Kudu in and of itself; however, it is fully supported to store data on disks that have been encrypted with full-disk encryption tools such as dm-crypt. Hence, anything read or written to these disks would be encrypted on the fly, and this would be transparent to the Kudu application itself.

Kerberos authentication

You enable Kerberos authentication by selecting [illegible] and Encryption" flag in Cloudera Manager. A few thing[illegible] cation to be enabled. A principal, `kudu@ACME.COM` [illegible]

```
--keytab_file=<path-to-keytab-file>
```

Enabling Kerberos through Cloudera Manager would create the principal and keytab file and set these automatically on the command-line start options of the Kudu services.

With this enabled, the Kudu service is now only allowing clients to connect who have been appropriately authenticated by Kerberos. However, note that any user who is authenticated by Kerberos can access the cluster at this point. Restricting who has access is what we describe in the next section.

User authorization

With Kerberos enabled, we can be sure that the client authenticating to our service is who they say they are, similar to our earlier analogy that the border guard has checked our passport and validated our identity. At the time of writing, Kudu on its own has two sets of users:

Superuser

: Administrative functionality to diagnose or repair issues using the `kudu` command-line tool.

User

: Access and modify all tables and data in a Kudu cluster.

When a distribution such as Cloudera manages the Kudu cluster, you can leave the Superuser parameter blank. This means that the service user that actually launched the Kudu master and tablet servers is allowed to do the administrative functions. In this case, it would simply be the user `kudu`, especially given that this is the principal name used for Kerberos. If it is managed by a distribution, you should leave this as is, unless there are specific reasons why a user needs to be able to use the comm[illegible] cluster for cases in which you cannot use Cloudera Manager d[illegible]

access might be restricted by firewalls, or perhaps we assign a single individual functional ID for Extract, Transform, and Load (ETL) jobs that is allowed to create, load, and manipulate content within Kudu.

Hence, we might only want two users given access to Kudu as a whole, which would include the following:

```
impala
etl_id
```

These parameters are set by the following command-line options, where we introduce the ID `mko_adm` as being a superuser administrator alongside the `kudu` ID defined by the service principal that started the kudu daemon:

```
--user_acl=impala,etl_id
--superuser-acl=kudu,mko_adm
```

This type of authorization is certainly coarse at the time of writing; however, back to the idea of Impala being the primary reporting or strategy in querying this table, all the authorization controls in place for Impala would apply. Impala integrates with Sentry for authorization, thus access to Impala tables, databases, and columns is controlled as it is capable of today.

We highlight the idea that it is important to remember the concept of an Impala table on top of Kudu is just that—they are decoupled. This means that although we are granted access to specific tables through Impala, if Kudu access through Kudu client API calls is not properly controlled by other means, users would be able to circumvent Impala altogether to manipulate and access data.

Log redaction

All row data is redacted by default from any Kudu s age -sitive data. There are two more options available to

You can set these by making the following adjustments:

```
--redact=log : No redaction in web UI, but server logs still are redacting
  sensitive data
--redact=none: No redaction performed anywhere (not recommended unless deb
```

Web UI security

Web UI security includes the ability to enable TLS for encrypting all traffic to and from clients connecting to the web UI. You need to prepare certificates in Privacy-Enhanced Mail (PEM) format, together with private keys, key passwords (password is emitted by running this specified command), and CA files, which you can set by using the following parameters:

```
--webserver-certificate-file=<path-to-cert-pem>
--webserver-private-key-file=<path-to-key-pem>
--webserver-private-key-password-cmd=<password-cmd>
```

Although TLS is enabled for encrypting traffic over the wire for the web UI, authorization for who is allowed access to this web UI is not controlled, (as of this writing) by Kerberos (which is typically done through SPNEGO).

Until access is restricted, you might decide to fully disable the web server itself by using this command:

```
--webserver-enabled=false
```

It is important to realize that doing this will also disable REST API endpoints that yield metrics for Kudu, which will have an impact on monito ave set up to monitor Kudu.

to keep in mind. Kudu is still in its infancy, but there are a few areas of performance tuning that as an administrator you should understand.

Thus far, a lot has been discussed about the type of underlying storage to make use of for the WALs and storage directories. In the chapters that follow, we take a closer look at schema design, which is absolutely critical to getting appropriate performance for your workloads.

From an administrator's perspective there are a few tunables to keep in mind that we list here, but we acknowledge that it is not an exhaustive list by any means. It is also very important to recognize when the performance bottleneck is actually on the server side of Kudu rather than the client. Sometimes, it is the clients that do not have enough resources (CPU or memory) or enough parallelism to drive the Kudu server to its limits.

From a high-level perspective, there are three primary areas of focus when it comes to tuning for performance:

- Amount of memory assigned to Kudu
- Appropriate partitioning strategies used (as mentioned, this is discussed in future chapters)
- The number of maintenance manager threads

Kudu Memory Limits

Kudu is built from the ground up in C++, using efficient and effective memory management techniques. Although it is meant to be lean, it is still worthwhile to provide Kudu with as much memory as you can allow in the cluster.

In multitenant clusters—and in this case, multitenant means various services all deployed in your big data environment such as HDFS, HBase, Impala, YARN, Spark—you might need to carefully design carving out memory appropri[illegible]

Here's the parameter to adjust:

In production environments, you should consider starting this value between 24 GB and 32 GB. You can certainly go higher for these memory settings as well.

Maintenance Manager Threads

Maintenance manager threads are background threads that perform various tasks. They do work such as performing memory management by specifically flushing data from memory to disk (thereby switching from the row-oriented in-memory format of records to a column-oriented on disk format), improving overall read performance or freeing up disk space. Work will be scheduled to these threads whenever a thread is made available.

> **NOTE**
>
> Row-oriented data is very easy to write out because very little processing is needed to write out a row. Row-oriented means that the data in that row has nothing to do with data in another row. For this reason, it is going to be a very fast write-oriented format. Column-oriented data is more difficult to write fast because it needs more processing to get a row into this format. As a result, column-oriented datasets are faster on reads. When Kudu holds data in memory, waiting for it to be flushed to disk, it is stored in write-optimized row format. However, after it is flushed to disk by the maintenance manager thread, it is then converted into columnar storage, optimized for aggregate read queries.

These are typically set according to the number of data drives that are assigned on a given tablet server. If there are 12 drives on a tablet server, we should set approximately four threads, or one-third the number of spinning disks. The faster the disks are, however, the more threads you can push. Starting from a number of one-third, it is safe to push this number in the range of the number of disks – 1. This is a general recommendation; you will need to test this for the given workload you are tuning. This is more to say do not leave this to the default of one thread, but also do not go into the many tens of threads set for this parameter.

Here's the parameter in question:

```
--maintenance_manager_num_threads
```

on certain performance metrics that might indicate a performance bottleneck.

JSON format metrics hook nicely into various dashboarding-type utilities and tools to provide insight as to what is going on in the system.

Getting Ahead and Staying Out of Trouble

As an administrator, it is a often a great idea to try to get ahead of having trouble with your cluster.

We make a few recommendations in this section to help guide you along.

Avoid Running Out of Disk Space

There are two parameters to keep in mind when trying to avoid running out of disk space. Any software platform may have undesired behavior, so there are a few parameters to stay ahead of the curve:

```
--fs_data_dirs_reserved_bytes
--fs_wal_dir_reserved_bytes
```

The default value for these is at 1% of disk space being reserved on the data and WAL directory filesystems for non-Kudu usage. Setting this close to the 5% range (though the parameter itself is set in number of bytes, not percentages, so you need to preform your own calculations) is one area to start with.

Disk Failures Tolerance

Kudu is a quickly evolving product, and is now resilient with sustaining disk failures occurring on tablet servers. Kudu master servers are not yet resilient to sustaining disk failures. As a result, when a disk fails, the entire Kudu master server is not usable.

Tablet servers are tolerant to disk failures; however, where the WALs or tablet metadata is stored, it will still incur

Tablet data will actually be striped across multiple disks that are assigned to a tablet server. The default value is 3, and if the value is set to the number of data directories or greater, data will be striped across all of the available disks on the system. If fewer data directories are assigned by tablet, there will be less chance of all tablets on a given tablet server being affected if a drive were to fail (given that data might not be striped on the disk for all tablets existing on the tablet server).

Following is the parameter that controls the number of data directories targeted for a given tablet replica:

```
--fs_target_data_dirs_per_tablet
```

Backup

Core backup functionality is still coming to Kudu, but even if it does shortly, it is still worth thinking about the various strategies that could be employed for backing up data.

Using MapReduce, Spark, or Impala, you could read Kudu tables and write them out to HDFS, preserving the schema but writing in Parquet format. After it is in HDFS, data can be shipped to other clusters in disaster recovery zones, or put in the cloud with the distcp utility, or even backed up with Cloudera's Backup Disaster Recovery (BDR) tool (which maintains table metadata as well among other benefits).

Another way to think about it is to ingest data immediately into two separate datacenters, especially if data is streaming in. Of course, you have to account for any outages in any of the Kudu clusters you're writing to, or even the speeds at which the tables are being populated because the two clusters might be out of synchronization as to how far they got in writing their data. Depending on the use case, this might be sufficient without much synchronization at all.

Conclusion

In this chapter, we covered some common administ… helped administrators plan their environments, discussed the web UI, and went through a

ing and tips on avoiding pitfalls. Next, we focus on the needs of the developer; namely, how to get up and running quickly as a developer of a new Kudu application using your programming language and framework of choice.

WATCH ON YOUR BIG SCREEN

View all O'Reilly videos, Superstream events, and Meet the Expert sessions on your home TV.