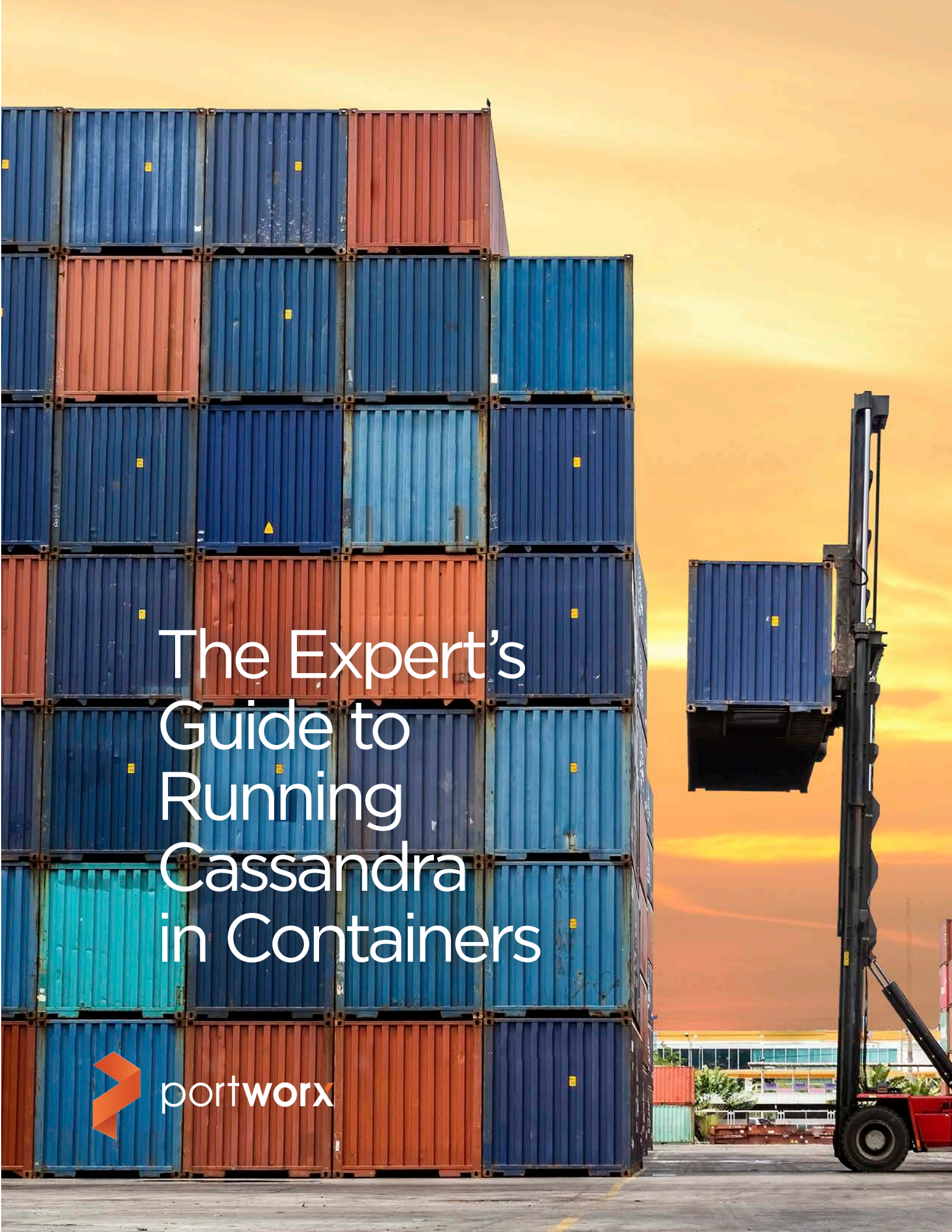


The Expert's Guide to Running Cassandra in Containers

portworx

Table of Contents

The Expert's Guide to Running Cassandra in Containers

Introduction

Apache Cassandra, first developed at Facebook, is an open-source distributed NoSQL database management system designed to handle large volumes of data across commodity servers. It is used at companies known for their large-scale operations, such as CERN and Instagram. As a result, Cassandra is responsible for both some of humanity's most awesome accomplishments and some of its most petty. Pretty good for an open-source project.

As a key part of microservice-based applications, Cassandra can benefit from being containerized. Because of this, it is one of the most popular images in the DockerHub, with over 5 million pulls.

This guide will present some best practices for running Cassandra in Docker containers with emphasis on:

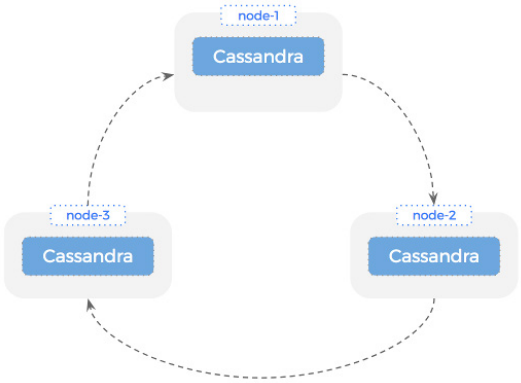
- Speeding up performance by colocating containers and their data on the same hosts
- Simplifying deployments using the Network Topology placement strategy
- Improving cluster recovery time after node failure
- Increasing container density by safely running multiple rings on the same hosts
- Achieving these benefits via your scheduler of choice: Kubernetes, DC/OS, or Swarm

Key Cassandra concepts

Before diving into discussing how to run Cassandra in Docker containers at scale, there are a few key concepts that we should define.

Ring

Cassandra's architecture is described as a "ring." This metaphor refers to how a single Cassandra database is automatically partitioned or sharded into multiple parts, which are then arranged into a metaphorical "ring" of nodes that make up the Cassandra cluster.



Partitioner

This partitioning is accomplished by a hashing mechanism based on the table's primary key. The piece of software that does the partitioning is called the...wait for it...partitioner. The key point here is that unlike manual sharding techniques based on, say, last name, where you might end up with some nodes with more data than others, the default partitioner randomizes data across the cluster, ensuring an even distribution of data.

Replication

One of the most popular features of Cassandra is its replication model, which unlike MongoDB or MySQL doesn't distinguish between a "primary" and a "copy." For each keyspace—a namespace that defines data replication on nodes—in your cluster, you can tell Cassandra how many copies to keep. This is called the replication factor. A replication factor of 1 would mean just one total replica of each partition, or in other words, no copies. With a replication factor of 1, you are living dangerously and highly at risk of data loss. Replication factor 2 would mean you make a single copy of each partition, so you have 2 total replicas. Replication factor 3 means you make 2 copies, so you have 3 total replicas. Replication factor 4 means...you get it.

Consistency

Very, very importantly, as mentioned above, each of these replicas is identical and can serve any read or write request. Cassandra has no notion of a master or slave node.

This is why Cassandra can get such good read and write performance. Just spin up a bunch of replicas, and you're good. They all keep track of what the others are doing using a protocol called "gossip."

Of course, nothing is free, and because writes can happen anywhere, keeping track of that activity across the cluster can be quite costly in terms of performance. But Cassandra has a concept of tunable consistency in which you can specify how many nodes you want acknowledging a write before telling the client application that it was successful (you can even tune consistency for reads, but here we'll focus just on writes).

The ALL consistency setting requires that a write must be written to all replica nodes in the cluster, and all of them must acknowledge the write before giving the client application the thumbs up. This will give you the highest consistency but the lowest availability (see CAP theorem for more about these tradeoffs). As a result, a common consistency setting is LOCAL_QUORUM, in which a write must be acknowledged by a quorum of nodes in a data center (e.g., 2 of 3 nodes, 3 of 5, etc.) before giving the thumbs up to the client application. Because you are waiting to receive acknowledgement of writes to fewer nodes than in the ALL consistency setting, you will have higher availability and probably good-enough consistency with the LOCAL_QUORUM setting.

Data placement strategies

Because we typically don't want our entire application to burn to the ground if we lose a single node, it matters where the replicas are placed.

Cassandra has two strategies for placing replicas to ensure HA:

SimpleStrategy: SimpleStrategy places the first replica in a ring on a node as determined by the partitioner. Other replicas are placed on the next nodes in the ring, moving clockwise without considering the physical location of your nodes, such as a rack or data center. This strategy is most often used for a single data center deployment. But because it does not ensure fault tolerance even among racks in the same data center, it is not recommended for production.

NetworkTopologyStrategy: NetworkTopologyStrategy places replicas in the same data center by walking the ring clockwise until reaching the first node in another rack. NetworkTopologyStrategy attempts to place replicas on distinct racks. That's because nodes in the same rack (or similar physical grouping) often fail at the same time due to power, cooling, or network issues. This strategy can also be used in multi-data-center deployments and requires specification on how many replicas are in each data center. Some common configurations are:

Two replicas in each data center: This configuration tolerates the failure of a single node per replication group and still allows local reads at a consistency level of ONE.

Three replicas in each data center: This configuration tolerates either the failure of one node per replication group at a strong consistency level of LOCAL_QUORUM or multiple node failures per data center using consistency level ONE.

Running Cassandra in Docker Containers

OK, with that background out of the way, let's look at how you can most successfully run Cassandra in a containerized environment.

Achieving top read/write performance with hyperconvergence

If you are running Cassandra, chances are that read/write performance is important to you. Cassandra was designed to run in bare-metal environments where each server offers its own disk as the storage media. The idea of running an app on the same machine as its storage is called "hyperconvergence." Cassandra will always get the best performance using this setup because it "uses disks heavily, specifically during writes, reads, compaction, repair (anti-entropy), and bootstrap operations"[1]—and shared storage puts pressure on these operations.

As Cassandra experts at DataStax also point out:

> What's important to understand about all of the disk I/O operations in Cassandra is that they are not only "heavy" in terms of IOPS (general rule of thumb is that Cassandra can consume 5,000 write operations per second per CPU code) but, arguably more importantly, they can also be very heavy in terms of throughput, i.e. MB/s. It is very conceivable that the disk system of a single node in Cassandra would have to maintain disk throughput of at least 200 MB/s or higher per node. Most shared-storage devices tout IOPS but don't highlight throughput as stringently. Cassandra will put both high IOPS as well as high throughput, depending on the use case, on disk systems. Heavy throughput is a major reason why almost all shared-storage Cassandra implementations experience performance issues.

Additionally, shared storage is a single point of failure that works against the resiliency built into Cassandra.

So direct-attached storage is the best approach, but in many data centers, this is not an obvious setup. In on-prem data centers, there is often a SAN available, but that is network-attached storage, not local storage. Many companies also have a Ceph or Gluster cluster that they try to use for centralized storage. Neither SAN nor Ceph/Gluster, however, delivers the performance that Cassandra likes. On the one hand,

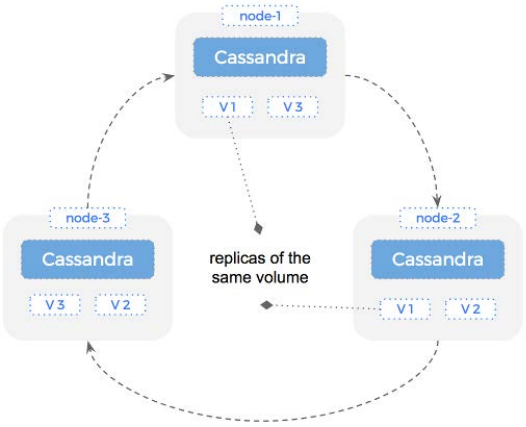
1 https://www.datastax.com/dev/blog/impact-of-shared-storage-on-apache-cassandra

the centralized nature of SAN storage increases the network latency and reduces throughput, to which Cassandra is sensitive. On the other hand, Ceph and Gluster, as file-based storage systems, are not optimized for database workloads, which again reduces performance.

In the cloud, we are usually talking about VMs, which besides not being bare metal are often backed onto an external block device such as Amazon EBS—which because it is network-attached, suffers the SAN-like latencies described previously that you don't get with direct storage media.

Portworx helps in both these situations, because it can force your scheduler—Kubernetes, Mesosphere DC/OS, or Docker Swarm—to run your container only on a host that already has a copy of the data.

For example, imagine you have a 3-ring Cassandra cluster with a replication factor of 2. That means you have your data sharded into 3 parts, and each part exists twice on the cluster.



3-node Cassandra cluster, replication factor 2

If one of your Cassandra containers crashes, say the container consuming V1 on Host 1, your scheduler will restart it. That's good. It's why we have a scheduler: to restart processes when they crash instead of waking us up.

However, without Portworx-enforced convergence, Kubernetes or DC/OS or Swarm might start that container up on Host 2 or 3, one that either does not have the data at all (Host 3) or that has a volume not already in use (Host 2). That means Cassandra will have to rebuild the hash from scratch, slowing down your entire cluster during the process.

But you already have the complete data volume on Host 1; the container has simply crashed. With Portworx-enforced scheduling constraints, you can automatically instruct your scheduler to start the container on a host with a free copy of V1. Because Host 1 is the only node that meets this criteria, it will be restarted in place.

Portworx enforces these types of scheduling decisions using host labels. By labeling hosts based on which volumes they have and passing these constraints to the scheduler, your containers will run in the location that minimizes rebuilds and maximizes performance using direct-attached storage.

Simplifying deployments using the Network Topology placement strategy

The previous section described how to automatically colocate containers and volumes on the same host, maximizing performance using direct-attached storage or, in the case of cloud, make sure you are running your containers on hosts with physical volumes such as EBS already attached.

While this setup helps with performance, it doesn't help with reliability. For that, we need to make sure that our Cassandra ring is deployed using the Network Topology strategy described earlier, so it is resistant to highly correlated failure modes such as rack or availability zone outages, cooling failures, or network partitions.

As a reminder, when placing replicas in a ring the Network Topology strategy takes into consideration what are called Fault Domains. A fault domain could be a host, a rack, or a data center. In contrast, the Simple strategy places replicas on nodes in your cluster without consideration of these things. The advantage of the Simple strategy is that it is simple. Its disadvantage is that availability will suffer.

The problem with the Network Topology strategy is that it is cumbersome to implement manually. Additionally, if you hand-place your containers in the optimum network topology, you can't take advantage of automated scheduling.

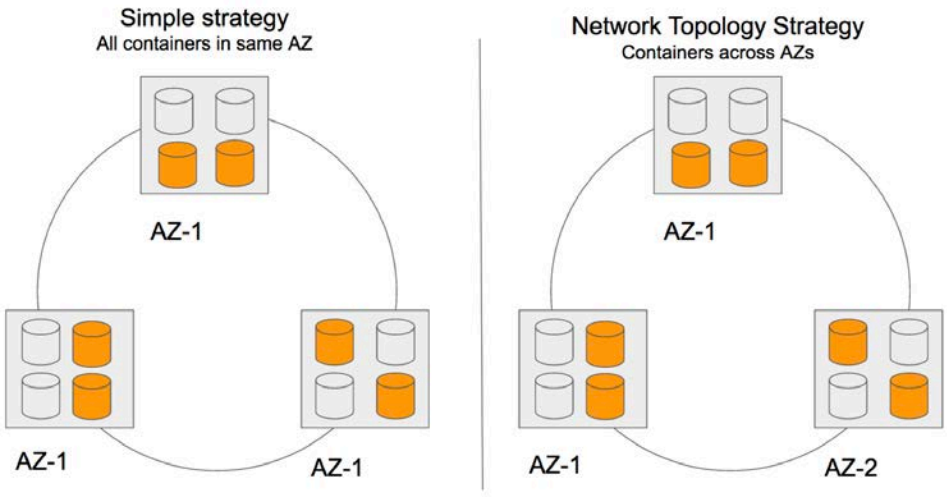
However, Portworx can automatically understand the topology of your data center and its fault domains and—as we've just seen—constrain scheduling decisions based on these factors.

In the cloud, Portworx automatically handles this placement by querying a REST API endpoint on the cloud providers for zone and region information associated with each host. With this understanding of your data center topology, Portworx can automatically place your partitions across fault domains, providing extra resilience against failures even before employing replication.

On prem, Portworx can also influence scheduling decisions by reading the topology defined in a yaml file such as cassandra-topology.properties. (Below, "DC" stands for data center and "RAC" stands for rack.)

Example:

```
# datacenter One

175.56.12.105=DC1:RAC1
175.50.13.200=DC1:RAC1
175.54.35.197=DC1:RAC1

120.53.24.101=DC1:RAC2
120.55.16.200=DC1:RAC2
120.57.102.103=DC1:RAC2

# datacenter Two

110.56.12.120=DC2:RAC1
110.50.13.201=DC2:RAC1
110.54.35.184=DC2:RAC1

50.33.23.120=DC2:RAC2
50.45.14.220=DC2:RAC2
50.17.10.203=DC2:RAC2

# Analytics Replication Group

172.106.12.120=DC3:RAC1
172.106.12.121=DC3:RAC1
172.106.12.122=DC3:RAC1

# default for unknown nodes default =DC3:RAC1
```



This feature is especially important when end users deploy Cassandra clusters themselves without knowledge of the data center topology. As enterprises increasingly make platforms available to their developers armed with a Kubernetes Helm chart, a Docker Compose file, or a DC/OS template, applications are increasingly deployed by people with little or no insight into the physical operating environment of that application. Automatic placement of replicas across fault domains takes care of this issue, without sacrificing availability.

Improving cluster recovery time after node failure

We've seen how you can deploy Cassandra replicas automatically in line with the Network Topology strategy and force a scheduler to run containers on hosts with named volumes.

Another consideration when running Cassandra in production is cluster recovery time in the event of outage or failure. Even with the Network Topology strategy, there will come a time when a replica needs to be rebuilt. This is usually due to a host failure.

Cassandra itself is capable of replicating data. If a node dies, a new node brought into the cluster will be populated with data from other healthy nodes, a process known as bootstrapping.

The bootstrap process puts load on your cluster, because cluster resources are used to stream data to new nodes. This load reduces read/write performance of Cassandra, which slows your application. For example, if you are using LOCAL_QUORUM consistency level, your application will have to wait longer because writes take longer to be acknowledged by a quorum of nodes. The key with bootstrapping is to finish as quickly as possible, yet this becomes difficult if the cluster is under stress.

Like Cassandra, Portworx also can bootstrap a node by replicating data from another host. Portworx replication, however, is at the block level. As a result, it is faster than Cassandra replication.

Block-level replication is faster for two main reasons:

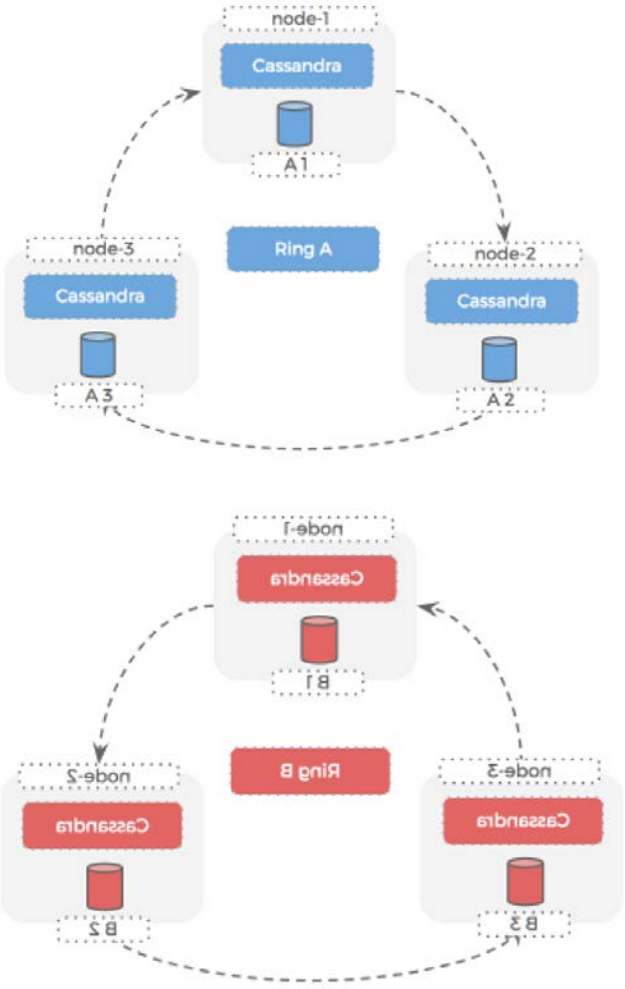
1. Block-level granularity allows for massive parallelization of network transfers so data is transferred faster.

2. Cassandra is written in Java, which is not as performant for the task of data transfer as C++—the language that Portworx is written in.

Increasing container density by safely running multiple rings on the same hosts

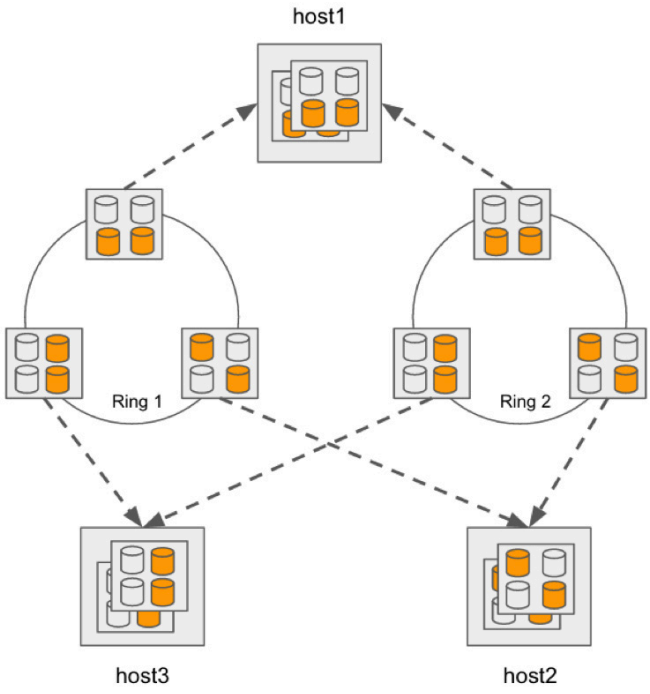
The previous operational best practices have been concerned with reliability and performance. Now we can look at efficiency. Most organizations run multiple Cassandra rings, and when each ring is architected as outlined earlier, you can achieve fast and reliable performance. However, because Cassandra is a resource-intensive application, the costs of operating multiple rings can be considerable. It would be nice if multiple rings could be run on the same hosts.

This is possible with Portworx. First, Portworx can provide container-granular volumes to multiple Cassandra replicas running on the same host. On prem, these volumes can use local direct-attached storage that Portworx formats as a block device and "slices" up for each container. Alternatively in the cloud, a single or multiple network-attached blocked device such as AWS EBS or Google Persistent disk can be used, again with Portworx slicing each block device into multiple container-granular block devices.



Then, the same placement algorithms described earlier apply to multiple rings. Because Portworx is application-aware, you could use the equivalent of a Cassandra ring ID as a group ID in volume. Using this ID, Portworx will make sure that is does not colocate data for two Cassandra instances that belong to the same ring in the same node. Each of your rings will be spread across the cluster, maximizing resiliency in the face of hardware failure.

In principle, it is possible to set up multiple Cassandra rings on the same hosts without a scheduler. This requires additional ring members to use different tcp ports for the Cassandra Docker container and update the corresponding csqshrc file. However, this option is complex to configure and moves teams away from the automation that speeds up deployments and reduces errors.



Both rings are running on the same hosts.

For more information, visit www.portworx.com.

Portworx, Inc.
4940 El Camino Real, Suite 200
Los Altos, CA 94022
Tel: 650-241-3222 | info@portworx.com | www.portworx.com