SOLUTION
DESIGN
GUIDE

DIAMANTI

crunchy data

SIMPLIFYING AND ACCELERATING CRUNCHY POSTGRESQL ON THE DIAMANTI ENTERPRISE KUBERNETES PLATFORM

Table of Contents

Running Databases at Scale with Kubernetes

Databases are a critical element of an enterprise IT environment. A significant percentage of IT budget is usually allocated to pay licensing fees for proprietary databases. Adopting a modern approach to deploying databases can make database operations much more agile by enabling Database-as-a-Service (DBaaS) and allowing database administrators to deploy database environments at scale.

In a recent Diamanti survey of IT decision makers, databases are one of the top two container use cases, chosen by 32 percent of respondents. Enterprises are seizing the opportunity to move away from expensive proprietary platforms in favor of open-source databases such as PostgreSQL that are well suited for container environments, scale faster, guarantee service levels, and cost less.

With containerized databases running on Kubernetes, developers and line-of-business teams can deploy a new database quickly without time consuming hardware configuration, and constant software installation and tuning. An important reason behind the widespread adoption of Kubernetes is that it helps organizations deploy applications using safe, scalable methods while providing a standardized management interface. Kubernetes lets applications run across different hardware platforms, so teams can choose where and how they want their workloads to be deployed and use the same procedures both on-premises and in the cloud.

Database management can require complex, multistep processes for provisioning, failover, backup/restore, and many other tasks. This document describes how to deploy PostgreSQL on Kubernetes using the PostgreSQL Kubernetes Operator from Crunchy Data and the Diamanti Enterprise Kubernetes Platform. This solution allows you to automate PostgreSQL management processes, streamline database deployment and management, and enable “PostgreSQL-as-a-service.” This document also describes how to use Diamanti Quality of Service (QoS) guarantees and Diamanti multi-zone availability and includes detailed performance analysis of the Diamanti platform for different simulated workloads in single and multi-tenant environments.

Solution Components

The solution described in this document consists of a number of different components:

- PostgreSQL
- Kubernetes
- Crunchy PostgreSQL
- Diamanti Enterprise Kubernetes Platform

This section describes each of these components in detail.

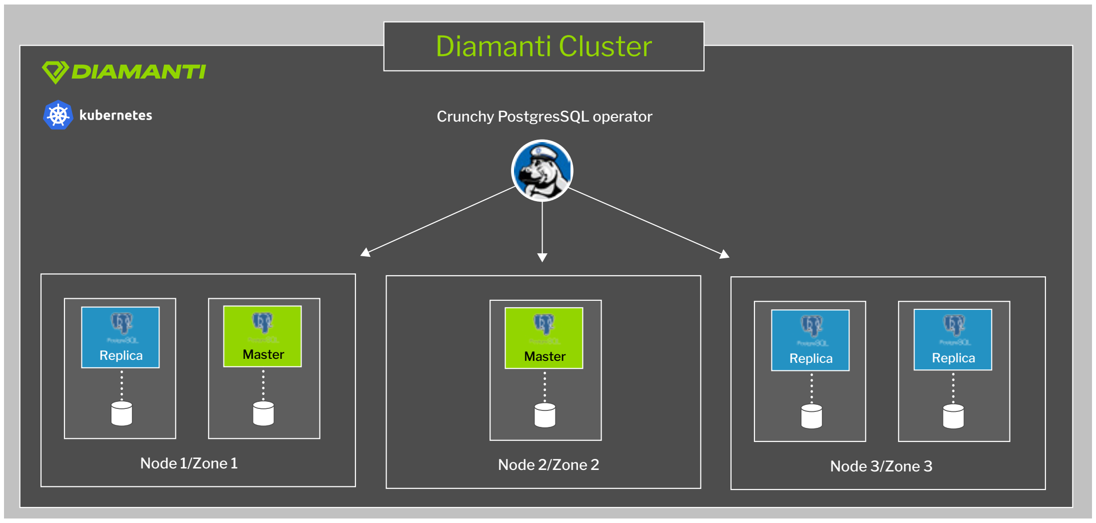


Figure 1: Logical relationship between components of this solution.

PostgreSQL

PostgreSQL is an open-source relational database management system (RDBMS) that has been under active development for more than thirty years. It is extremely popular due to its robust feature set for application development as well as highly reputed for data integrity and performance.

As with any database system, running PostgreSQL in a mission-critical production environment requires careful management to ensure key tasks are carried out correctly. For instance, in a high-availability environment, successful failover from a primary to a replica requires execution of the following steps in sequence:

- Hold all connections from continuing to perform transactions on the primary
- Shut the primary down and ensure it cannot come back up
- Determine which replica is the furthest along in the timeline, i.e. which replica is closest to the state of the primary at the time of the failover
- Choose the best replica and promote it to become the primary
- Point the remaining replicas at the new primary so they can read in the remaining transactions
- Allow connections to resume transacting with the new primary
- Re-provision a new replica to return the appropriate balance to the HA cluster

Detailed, multistep processes like this one can be difficult to get right, making automation essential when operating at scale.

Containers and Kubernetes

Many organizations are seeking faster and more flexible methods for deploying and managing databases and applications. To address issues of portability, efficiency, and scale, many organizations are turning to containers and Kubernetes. Since it was open-sourced in 2014, Kubernetes has emerged as the leading platform for managing containerized applications.

An important reason behind the widespread adoption of Kubernetes is that it helps organizations deploy applications using safe and scalable methods while providing a standardized management interface. Kubernetes lets applications run across different hardware platforms, so teams can choose where and how they want workloads to be deployed and use the same procedures both on premises and in public clouds.

In database environments, it's imperative that administrators follow approved processes for management tasks including:

- Provisioning a new database cluster
- Failing over a cluster to a new primary server
- Restoring data from a backup

Manual task execution simply becomes too error prone and too time consuming for operations that run hundreds, if not thousands, of database applications.

This is where the concept of a Kubernetes Operator comes in. An Operator gives one-click interface to database services so that essential functionality which, is necessary to manage the databases, can be performed quickly, reliably and predictably.

Kubernetes Operators

Operators make use of Kubernetes **Custom Resource Definitions (CRDs)** to define specifics about a database application (such as which storage to use) and can regulate the overall health of the services they are managing.

Operator commands can be applied to individual instances of the database applications they are managing or across an entire database cluster. This allows you to provide Operators to teams with standardized commands that capture the different nuances for a particular application; there is less friction for people who need to do development work for an application, but do not necessarily have the in-depth expertise to manage it.

A well-designed Operator can give you the equivalent of an 'as-a-service" technology that you can deploy anywhere you have a Kubernetes cluster.

Crunchy PostgreSQL

Crunchy Data was founded in 2012 with the mission of bringing the power and efficiency of open-source PostgreSQL to security-conscious organizations and eliminating expensive proprietary software costs.
The Crunchy PostgreSQL Operator encapsulates key PostgreSQL-as-a-Service operations to manage the full lifecycle of PostgreSQL, including:

- Database provisioning and deletion
- High-availability
- Data protection and disaster recovery
- Cloning
- User management
- Labeling and grouping for policy management

The *Crunchy PostgreSQL Operator 3.4* introduced the ability to manage databases across multiple Kubernetes namespaces from a single Operator to create additional isolation and security for Kubernetes clusters being used by multiple tenants.

The Crunchy PostgreSQL Operator provides a straightforward command-line interface (CLI) for all supported operations. As an example, the following command creates a new database:

```
$ pgo create cluster newcluster
```

While the Crunchy PostgreSQL Operator standardizes workflows, it allows for complete customization using Kubernetes Deployments.

The Crunchy PostgreSQL Operator allows you to flexibly manage a production PostgreSQL cluster from a single Operator instance. It includes features that allow an administrator to:

- **Mix different storage classes.** Specify which storage type each PostgreSQL instance, replica or backup should preferentially utilize.
- **Specify node affinity.** Choose which node(s)/compute resources a PostgreSQL instance should preferentially utilize.
- **Mix different resource profiles.** Specify resource profiles for memory and CPU allocation on a per-instance or per-tenant basis.
- **Run different versions.** Run newer applications on PostgreSQL 11 while running older applications on PostgreSQL 10.
- **Use enterprise user access controls.** Integrate PostgreSQL access control with existing enterprise access control systems.

Diamanti Enterprise Kubernetes Platform

When running a PostgreSQL database cluster, *the performance and availability characteristics of the Kubernetes platform are critical.* High-performance PostgreSQL databases depend on network and storage I/O and should run on the fastest storage media possible, with the ability to allocate different performance levels to different database instances. The Kubernetes platform must be highly resilient with no single points of failure.

You have to make careful infrastructure choices to avoid being saddled with a solution that is overly complex and difficult to manage, lacks the necessary performance, or locks you into a specific vendor or cloud environment. The Diamanti Enterprise Kubernetes Platform provides a hyperconverged infrastructure with Docker and Kubernetes fully integrated to allow developers and administrators to easily run applications. Storage and networking quality of service (QoS) deliver the performance necessary for demanding production database environments.

Diamanti Enterprise Kubernetes Platform is Kubernetes certified. The platform provides storage via a standard container storage interface (CSI) plugin and networking via a standard container network interface (CNI) plugin, delivering low-latency access to both storage and network resources on bare-metal appliance using hardware offload and acceleration. Built-in HA eliminates single points of failure, and single-click simplicity makes it easy to add resources on demand.

Table 1: Why Run PostgreSQL on the Diamanti Enterprise Kubernetes Platform?

FEATURE	BENEFITS
Quality of Service (QoS)	▪ Eliminates noisy neighbors ▪ Provides isolation at the PCIe layer ▪ Delivers guaranteed performance ▪ Integrated with k8s storage class ▪ Customizable performance tiers

Table 1: Why Run PostgreSQL on the Diamanti Enterprise Kubernetes Platform?

FEATURE	BENEFITS
Performance	■ Fast NVMe storage with million+ IOPS ■ Sub-millisecond latency across the cluster (at full load)
Hardware-level replication	■ Diamanti Mirroring (across nodes) ■ Diamanti Backup Controller for snapshot-based backups ■ Separate storage network (for NVMoE traffic) ■ Quick recovery of pods in case of failure
Simplified L2 networking	■ Direct TCP connectivity to pods via data network ■ No need to route traffic via host/nodes ■ Separate control and data planes ■ Support for network Endpoints (static IP address) ■ Easy communication between replicas
Multi-zone (AZ) support for better HA and DR	■ Support stretched/campus clusters ■ Schedule pods with storage across multiple zones for HA

DEPLOYING CRUNCHY POSTGRESQL OPERATOR 3.4 ON DIAMANTI

The Crunchy PostgreSQL Operator provides a single management plane for all the PostgreSQL database instances running on a Diamanti cluster. This section describes how to deploy version 3.4 of the Operator. This section assumes that you already have the Diamanti Enterprise Kubernetes Platform deployed. If you need information on configuring and deploying Diamanti hardware and software, refer to Diamanti user's guide.

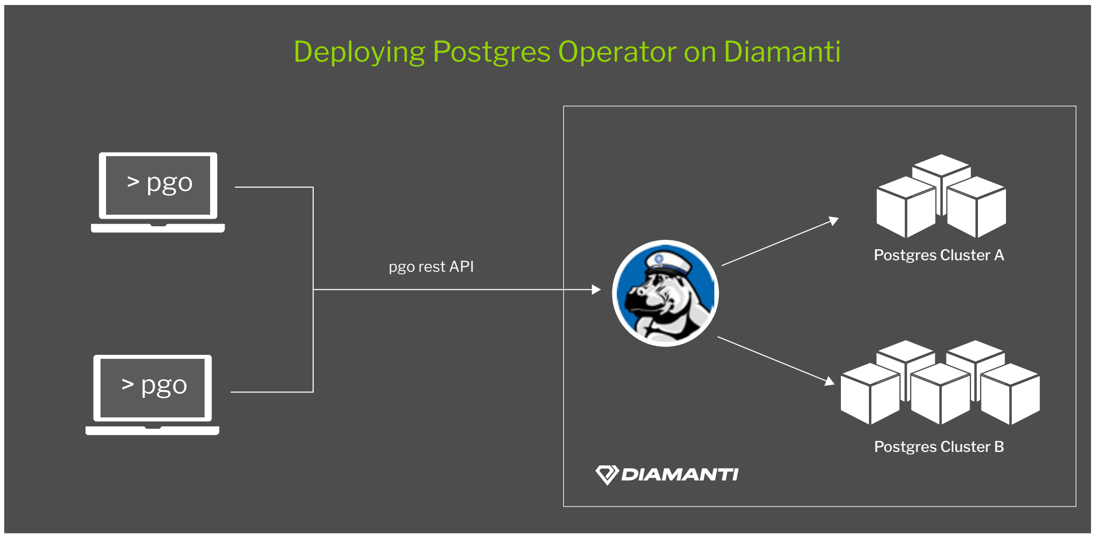


Figure 2: Deploying the PostgreSQL Operator on the Diamanti Enterprise Kubernetes Platform.

Once you deploy the Operator on a Diamanti cluster, the Kubernetes software ensures that the Operator is always running. Users can connect to the PostgreSQL Operator REST API via the pgo client or create their own user interface to manage databases running on the cluster via the Operator. The Operator allows users to perform various management operations based on access controls; this includes cluster creation, backup, restore, failover, and monitoring.

The Crunchy PostgreSQL Operator lets you create PostgreSQL clusters customized for the needs of your organization/users. Operator capabilities include:

- Create simple single-instance clusters
- Create highly available clusters with multiple read replicas that can take over in case of failures
- Scale read replicas dynamically when load is growing or shrinking
- Add pgpool as an interface to a database for connection pooling and read/write balancing
- Schedule database backups using pgbackrest or pgbase backup
- Easily restore PostgreSQL instances from backup
- Easily upgrade PostgreSQL clusters online with no downtime

More details about the Crunchy PostgreSQL Operator can be found at https://crunchydata.github.io/postgres-operator/stable/

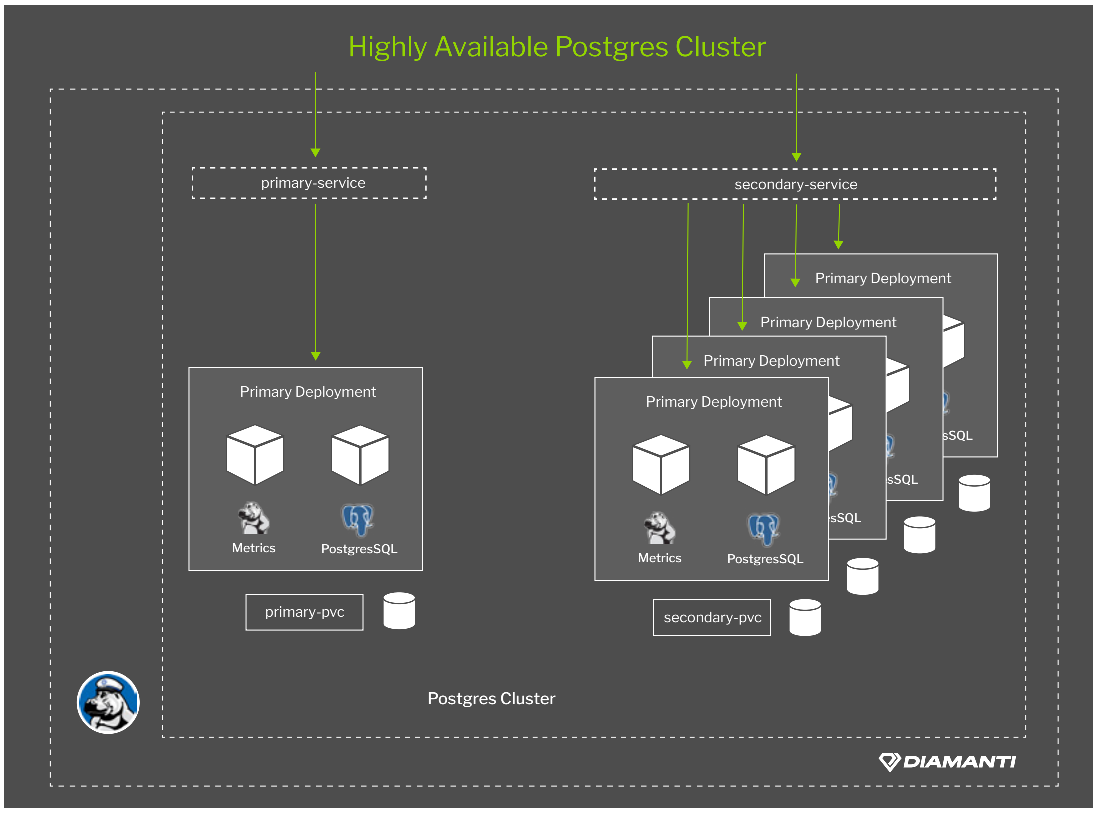


Figure 3: The Crunchy PostgreSQL Operator simplifies enabling high availability.

Deploying and Using the Crunchy PostgreSQL Operator

The simplest way to deploy the Crunchy PostgreSQL Operator (postgres-operator) is using a Helm chart. (Helm is a popular package manager for Kubernetes.) You can download the necessary Helm chart and other required tools from:

https://github.com/CrunchyData/postgres-operator/releases/tag/3.4.0

Once the tools are installed, enter the following command:

```
$ helm install chart/postgres-operator -n pg-operator
```

This will deploy the necessary postgres-operator pod. The IP address of the Operator pod must be used to configure the **pgo client**. Installation of the postgres-operator deploys various new CRDs in the Kubernetes cluster, extending Kubernetes APIs to incorporate new objects as shown in the following command output.

```
$ kubectl get crd | grep pg
pgbackups.cr.client-go.k8s.io                                      5s
pgclusters.cr.client-go.k8s.io                                     5s
pgingests.cr.client-go.k8s.io                                      5s
pgpolicies.cr.client-go.k8s.io                                     5s
pgreplicas.cr.client-go.k8s.io                                     5s
pgtasks.cr.client-go.k8s.io                                        5s
pgupgrades.cr.client-go.k8s.io                                     5s
```

Deploying PostgreSQL Clusters

Once the postgres-operator has been deployed and the local ***pgo client*** configured, you are ready to deploy postgreSQL clusters on Diamanti. The following examples illustrate some of the most common commands:

Create a single-instance cluster:

```
$ pgo create cluster pg-cluster-single
```

Create a highly available cluster:

```
$ pgo create cluster pg-cluster-ha --replica-count=3
```

Scale an existing cluster:

```
$ pgo scale pg-cluster-ha --replica-count=2
```

View all users and associated passwords for a specific cluster:

```
$ pgo show user pg-cluster-single
```

View details of a specific cluster:

```
$ pgo show cluster pg-cluster-single
```

The PostgreSQL Operator greatly simplifies the deployment, monitoring, and scaling of database clusters.

Additional Cluster Configuration Options

The Crunchy PostgreSQL Operator includes a number of additional options that allow you to tailor your PostgreSQL environment to your needs. These options can be used individually or in combination as appropriate.

AUTO-FAILOVER

When auto-failover is enabled, the postgres-operator automatically promotes a slave replica to become the master should the master fail. The old master is deleted and a new slave replica is created.

```
$ pgo create cluster pg-cluster-ha-af --replica-count=2 --autofail
created Pgcluster pg-cluster-ha-af
workflow id 0c87e0a1-4020-4660-85e6-da212d7119af
```

CAPTURING METRICS

The following command creates a cluster with metrics enabled to allow you to monitor database instances:

```
$ pgo create cluster pg-cluster-metrics --metrics
created Pgcluster pg-cluster-metrics
workflow id 1abb5b19-1f61-409e-8bde-3e6694a22011
```

PREDEFINED CPU AND MEMORY

You can create PostgreSQL clusters with predefined resource configurations to control how much CPU and Memory PostgreSQL instances can consume as illustrated in the following command:

```
$ pgo create cluster pg-cluster-xlarge --replica-count=3 --resources-config=d10-xlarge
created Pgcluster pg-cluster-xlarge
workflow id defe0e6e-a8a0-4ab0-a8f3-a1a238e1f219
```

In the above example, d10-xlarge is a custom resource predefined in Postgres Operator configuration file (pgo.yaml) which was used when deploying the pgo operator.

STORAGE CONFIGURATION

The Crunchy PostgreSQL Operator also allows you to define the Diamanti storage configuration for both masters and replicas as shown in the following command.

```
$ pgo create cluster pg-cluster-high --storage-config=d10-high --replica-storage-
config=d10-medium
created Pgcluster pg-cluster-high
workflow id 03494707-5363-4cb9-a02f-fa1fbea469da
```

In the above example, d10-xlarge and d10-medium are custom resources predefined in configuration file (pgo.yaml) which was used to deploy the pgo operator.

Lifecycle Management with the PostgreSQL Operator

Not only does the Crunchy PostgreSQL Operator simplify the deployment of PostgreSQL instances, it extends the APIs and CLI to help in day-to-day PostgreSQL cluster operations. The following commands illustrate some of the most common operations:

BACKUP AND RESTORE

Create a backup of a specific cluster:

```
$ pgo backup pg-cluster-single
created Pgbackup pg-cluster-single
```

Display information about a previously created backup:

```
$ pgo show backup pg-cluster-single
pgbackup : pg-cluster-single
        PVC Name:       pg-cluster-single-backup
        Access Mode:    ReadWriteOnce
        PVC Size:       40G
        Creation:       2019-04-03 15:03:53 -0700 PDT
        CCPImageTag:    centos7-10.6-2.2.0
        Backup Status: completed
        Backup Host:    pg-cluster-single
        Backup User Secret:    pg-cluster-single-primaryuser-secret
        Backup Port:    5432
```

Restore a cluster from a previous backup:

```
$ pgo create cluster pg-cluster-restored --backup-path=/pgdata --backup-pvc=pg-cluster-
single-backup --secret-from=pg-cluster-single
created Pgcluster pg-cluster-restored
workflow id 8e8b02ba-3962-49b7-baf5-6369e032ddb2

$ kubectl get pod --selector=pg-cluster=pg-cluster-restored -o wide
NAME                                    READY   STATUS    RESTARTS   AGE    IP              NODE
pg-cluster-restored-5c69fb6c98-5hbdb    1/1     Running   0          0s     172.16.225.61   solserv9
```

UPGRADE AND ROLLBACK

The Operator also makes it simple for you to upgrade a running PostgreSQL cluster to a new software version. The commands shown below allow you to:

- See the currently deployed image tag
- Upgrade the cluster with a new container image
- Verify the new image is deployed

```
$ kubectl get pods  -l pg-cluster=pg-cluster-single -o jsonpath=
'{.items..spec.containers[?(@.name == "database")].image}'
crunchydata/crunchy-postgres:centos7-10.6-2.2.0

$ pgo upgrade pg-cluster-single --ccp-image-tag centos7-10.6-2.3.0
created Pgupgrade pg-cluster-single
$ kubectl get pods  -l pg-cluster=pg-cluster-single -o
jsonpath='{.items..spec.containers[?(@.name == "database")].image}'
crunchydata/crunchy-postgres:centos7-10.6-2.3.0
```

Applying Diamanti QoS

The Diamanti Enterprise Kubernetes Platform uses QoS to eliminate the noisy neighbor problems that can affect container deployments. Diamanti lets you run secondary database workloads like backups without impacting primary database workloads.

We ran an experiment to measure the impact of backup jobs on active databases. The **pgbench** workload was run against a single-instance PostgreSQL database cluster, and a backup of the database was created while the benchmark was running. Two performance tiers were configured on Diamanti platform: high and best-effort. The database cluster was assigned the high-performance tier while the backup job was assigned the best-effort-performance tier. The **pgbench** results were not impacted by the backup job.

The following commands were used to perform the test:

Create a Postgres cluster:

```
$ pgo create cluster pg-cluster-0 --replica-count=0 --resources-config=xlarge
created Pgbackup pg-cluster-0
workflow id 34cbf32c-23bf-4c5c-ae54-baa8fd779b12
$ kubectl get pods -l pg-cluster=pg-cluster-0,primary=true -n default -o wide
NAME                           READY    STATUS    RESTARTS   AGE    IP              NODE
pg-cluster-0-597746749c-b6g7v  1/1      Running   0          1m     172.16.225.7    solserv7
```

Run the benchmark (Select-only query workload):

```
$ pgbench -r -S --host=172.16.225.7 --port=5432 --username=primaryuser --jobs=2
--time=300 --client=16 pgbench
```

Run backup while the benchmark is running:

```
$ pgo backup pg-cluster-0
created Pgbackup pg-cluster-0
```

The results are summarized in the following screenshot and output. Note the backup has no negative impact on benchmark performance.

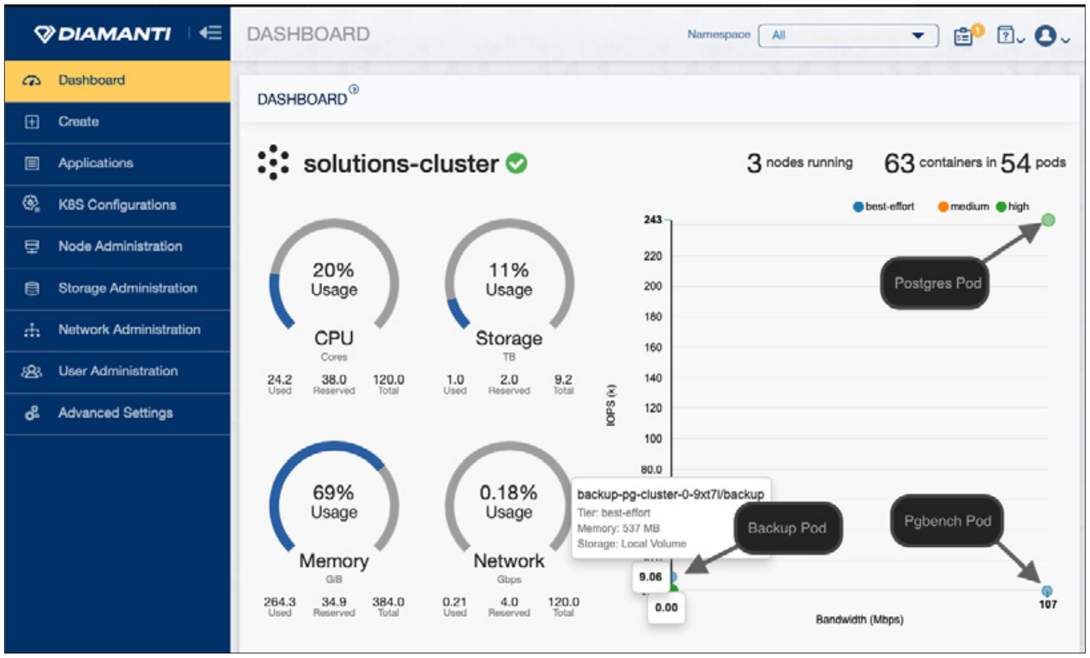


Figure 4: Load on the Diamanti Enterprise Kubernetes Platform while running the pgbench benchmark with select-only query workload and a simultaneous backup.

Benchmark latency and TPS with no backup:

```
TRANSACTION TYPE: <BUILTIN: SELECT ONLY>
... OUTPUT OMITTED ...
LATENCY AVERAGE = 0.372 MS
TPS = 42955.836795 (INCLUDING CONNECTIONS ESTABLISHING)
TPS = 42956.326779 (EXCLUDING CONNECTIONS ESTABLISHING)
... OUTPUT OMITTED ...
```

Benchmark latency and TPS with backup running:

```
TRANSACTION TYPE: <BUILTIN: SELECT ONLY>
... OUTPUT OMITTED ...
LATENCY AVERAGE = 0.356 MS
TPS = 44904.729067 (INCLUDING CONNECTIONS ESTABLISHING)
TPS = 44905.232620 (EXCLUDING CONNECTIONS ESTABLISHING)
... OUTPUT OMITTED ...
```

A second **pgpbench** test used a simulated TPC-B workload.

Run the benchmark (TPC-B like workload):

```
$ pgbench -r --host=172.16.225.7 --port=5432 --username=primaryuser --jobs=2
--time=300 --client=16 pgbench
```

Run backup while the benchmark is running:

```
$ pgo backup pg-cluster-0
CREATED PGBACKUP PG-CLUSTER-0
```

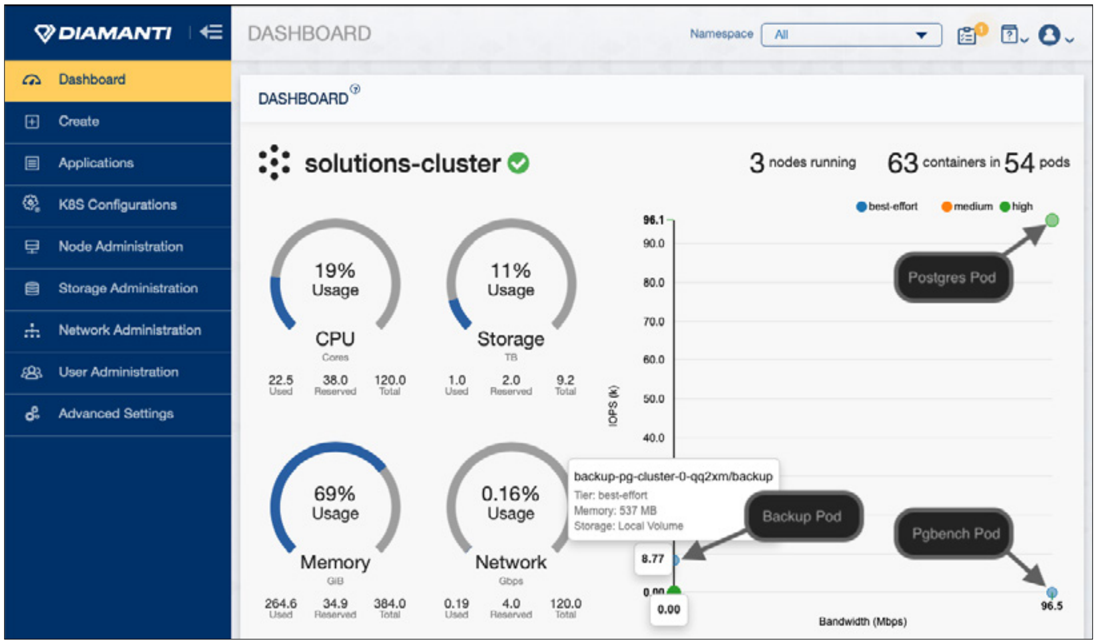


Figure 5: Load on the Diamanti Enterprise Kubernetes Platform while running the pgbench benchmark with simulated TPC-B workload and a simultaneous backup.

Benchmark latency and TPS with no backup:

```
TRANSACTION TYPE: <BUILTIN: TPC-B (SORT OF)>
... OUTPUT OMITTED ...
LATENCY AVERAGE = 2.208 MS
TPS = 7247.100662 (INCLUDING CONNECTIONS ESTABLISHING)
TPS = 7247.179264 (EXCLUDING CONNECTIONS ESTABLISHING)
... OUTPUT OMITTED ...
```

Benchmark latency and TPS with backup running:

```
TRANSACTION TYPE: <BUILTIN: TPC-B (SORT OF)>
... OUTPUT OMITTED …
LATENCY AVERAGE = 2.289 MS
TPS = 6990.571330 (INCLUDING CONNECTIONS ESTABLISHING)
TPS = 6990.653528 (EXCLUDING CONNECTIONS ESTABLISHING)
... OUTPUT OMITTED …
```

The changes in TPS and latency numbers are attributed to the way the backup is performed, adding an rsync process to the PostgreSQL cluster. This is especially true for the simulated TPC-B workload. The slight increase in TPS numbers for the select query in the first example may be attributable to a decrease in cache misses. Only one database instance was running in the cluster, and it had enough CPU resources. In an oversubscribed multi-tenant system, you should expect greater variance due to the backup. The architecture for the Crunchy backup method is shown in Figure 6.

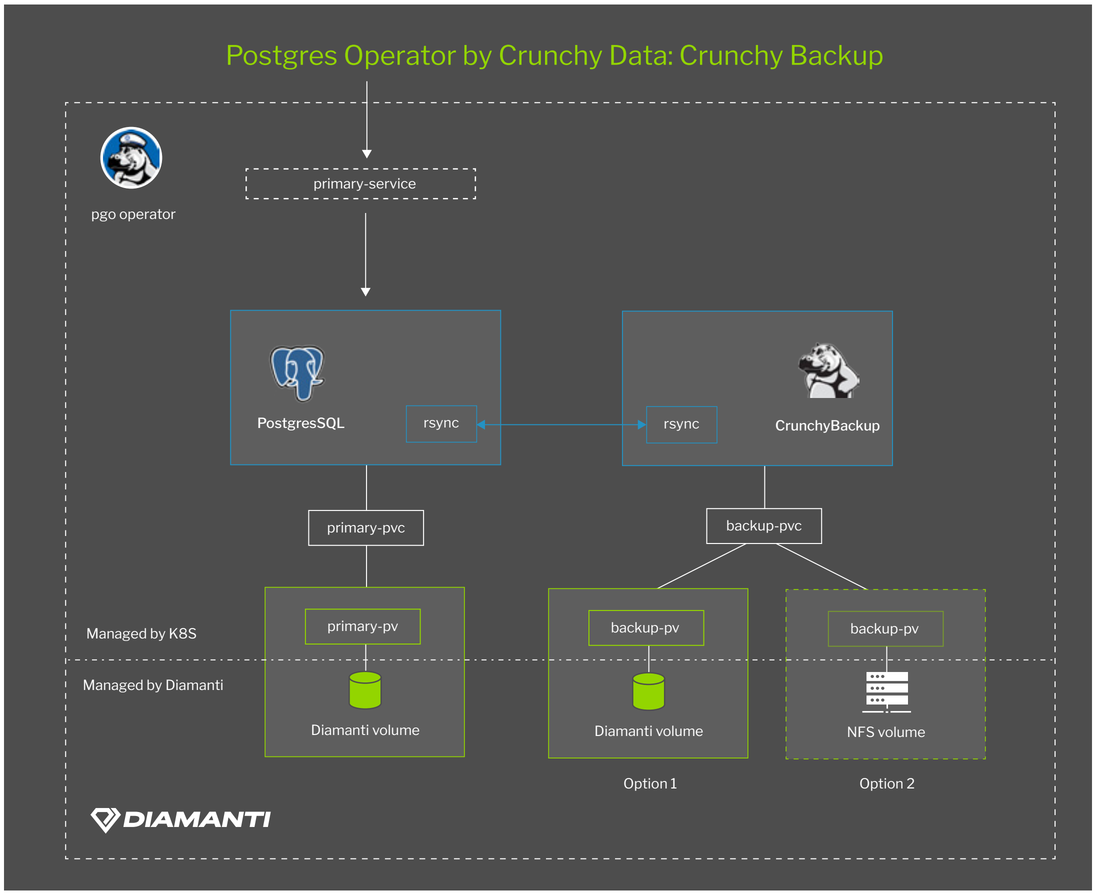


Figure 6: The PostgreSQL Operator uses rsync to create backups to Diamanti persistent volume(s).

There is an alternative method to create backups ***with total isolation of resources*** using Diamanti storage services. This approach takes advantage of instant, space-optimized snapshots using Diamanti's backup CRD. This method is described later in the section under Diamanti storage services.

Multi-Zone Deployments for High Availability

The Diamanti Enterprise Kubernetes Platform supports Availability Zones (AZ) from both a network and storage standpoint. Availability Zones can be defined to mean three different racks within a single data center or three different data centers within the same region, where a region is defined to allow latency below 2ms. In practice, this often means a dark fiber (DWDM) link between data centers with reasonable geographic proximity. To support NVMe over ethernet storage traffic, the storage VLAN must be stretched across data centers.

Figure 7 illustrates two different approaches to create highly available PostgreSQL deployments with instances distributed across Availability Zones. One uses application-level replication while the other makes use of Diamanti mirroring. Diamanti storage scheduling is AZ aware and, in both approaches, ensures that the persistent volumes required for the PostgreSQL pods are distributed across AZs to ensure no single point of failure.

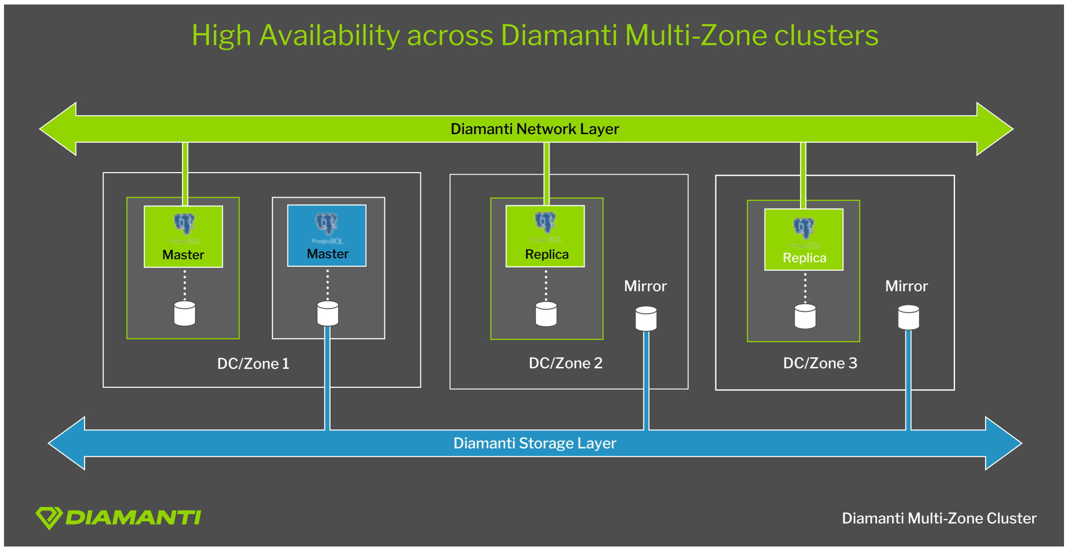


Figure 7: High availability can be enabled using application-level or storage-level replication.

Approach 1: Application-level Replication and HA with PostgreSQL

In this approach, the replication of data is handled by PostgreSQL. The master instance uses the network to replicate the PostgreSQL Write Ahead Log (WAL). The replica instances are up and running, which means that they consume CPU, RAM, storage, and network resources. The advantage of this approach is that replica instances can serve read requests should the load increase, providing read scaling. In addition, if the master (Zone 1) fails, one of the replicas will become the master. For this type of configuration there has to be a load balancer (for example **pgpool**) associated with the deployment.

Approach 2: Storage-level Replication and HA with Diamanti

In this approach, the Diamanti storage layer provides redundancy for the persistent volume associated with the master instance. This is achieved by defining a storage class which specifies the number of mirrors (three in the figure). As illustrated, the Diamanti storage layer does the mirroring via the storage VLAN, segmenting network and storage traffic.

No system resources are consumed in Zone 2 and Zone 3, and no read scaling is provided. If the Master (Zone 1) fails, a master instance will be started in one of the other zones (Zone 2 or Zone 3). The time to failover in both approaches is the same. For approach 2, it is important that the new instance have the same IP address as the original master. For this reason, a network "endpoint" is used in this type of deployment.

DIAMANTI STORAGE SERVICES

For backups, you can also take advantage of Diamanti's backup CRD (Custom Resource Definition), based on zero-copy, instant snapshots. This method provides complete isolation of the primary PostgreSQL service in terms of system resources (CPU/RAM). With Diamanti QoS, even when the **backup-pv** shares the same data blocks as the **primary-pv**; there is no impact on performance of the primary PostgreSQL database.

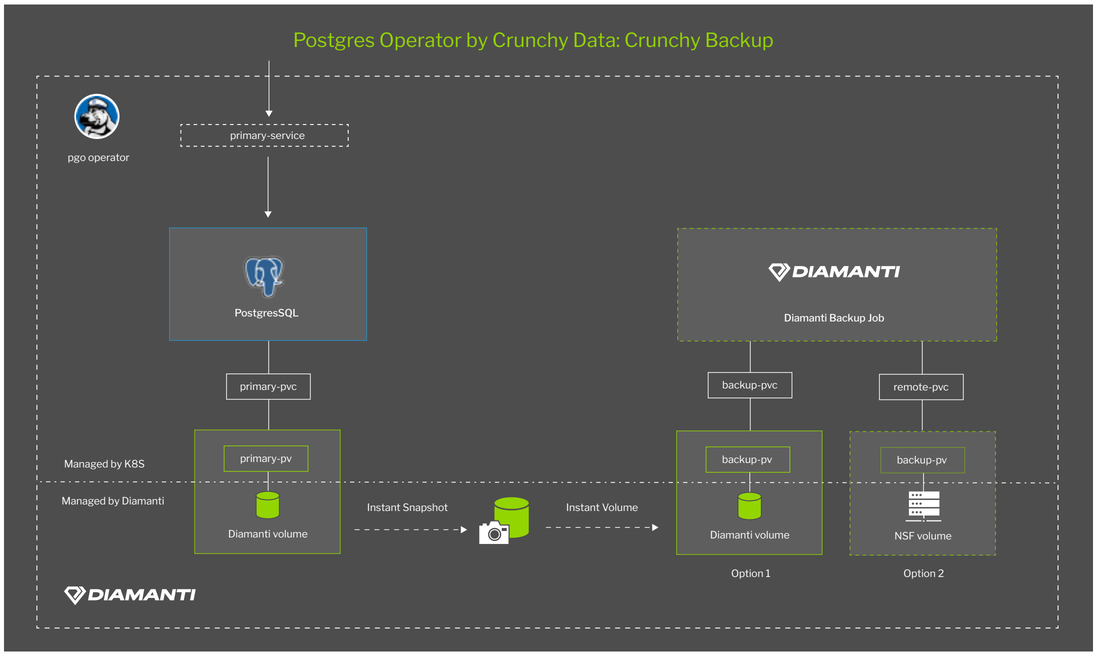


Figure 8: Backups can be enabled using Diamanti Storage Services, offloading the workload from PostgreSQL.

Pushing the Diamanti Platform Limits

In this section we explore the limits of a 3-node Diamanti cluster. We tested using 63 PostgreSQL instances per node for a total of 189 PostgreSQL instances. Once again, **pgbench** was used to drive load across all PostgreSQL instances.

Diamanti Hardware Configuration[1]

- CPUs: 64 HT Cores
- Memory: 256 GB of RAM
- Storage: 3 TB
- VNICs: 63

```
$ dctl cluster status
... Output Omitted ...
NAME         STATE   CONDITION   MILLICORES    MEMORY           STORAGE     IOPS      VNICS

BANDWIDTH    STORAGE-CONTROLLERS


TOTAL, REMOTE

APPSERV91   READY   READY       0/64000      245.7G/251.63G  1.23T/2.78T  315K/500K        63/63

7.88G/40G               63/64, 0/64

APPSERV92   READY   READY       0/64000      245.7G/251.63G  1.23T/2.78T  315K/500K        63/63

7.88G/40G               63/64, 0/64

APPSERV93   READY   READY       0/64000      245.7G/251.63G  1.23T/2.78T  315K/500K        63/63

7.88G/40G               63/64, 0/64
```

[1] *The Diamanti platform is available in several hardware configurations. This testing was performed with one such configuration which is D10.*

The Diamanti platform integrates storage and network resources into the cluster. The scheduler extensions allow Diamanti to schedule pods while taking into consideration the network resources (bandwidth and number of interfaces) and storage resources (capacity and IOPS) in the cluster. Any other Kubernetes distribution, especially with multi-vendor plugins for CSI (storage) and CNI (network) would not be able to integrate these capabilities into the cluster.

Test Configuration

The goal of this exercise is to consume all system resources; so we deployed PostgreSQL pods with the following configuration:

- Single instance PostgreSQL pods
- 63 PostgreSQL instances per node

Resource consumption per pod:

- Image: single-instance PostgreSQL
- CPU: 1 HT
- RAM: 3.9 GB
- Volume size: 20 GB
- Data load factor: 600
- Capacity used: 10–12 GB

Resource consumption per node:

- Memory: Each pod used 3.9 GB RAM, which resulted in use of 250 GB+ RAM per node
- QoS: We used the medium QoS performance tier for all pods, which guarantees 5k IOPS per pod for a total of 63 × 5 = 315k IOPS provisioned for each node

Resource consumption (cluster):

- Number of nodes: 3
- Total pods: 189
- QoS: 945k provisioned IOPS

We created 60M records and 12 load clients with 2 query threads each per instance of the PostgreSQL database.

Performance Results: Select-Only Queries

The **pgbench** load configuration is shown below:

```
transaction type: <builtin: select only>   → select only queries
scaling factor: 600                         → creates 60M tuples
query mode: simple
number of clients: 12                      → Load from Client
number of threads: 2
```

We ran **pgbench** as described above and monitored the results via the Diamanti user interface and pgbench output. The screenshot in Figure 9 shows the IOPS/bandwidth curve with the pgbench load running. Each pod consistently achieves approximately 15K average IOPS. Although the minimum guaranteed IOPS was 5k per pod, the extra IOPS available on the system were evenly distributed across all pods.

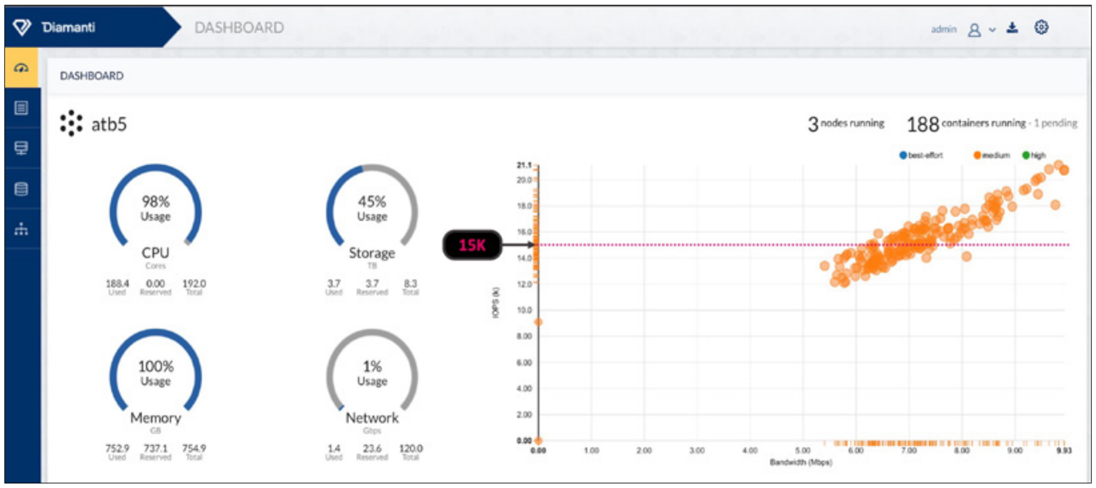
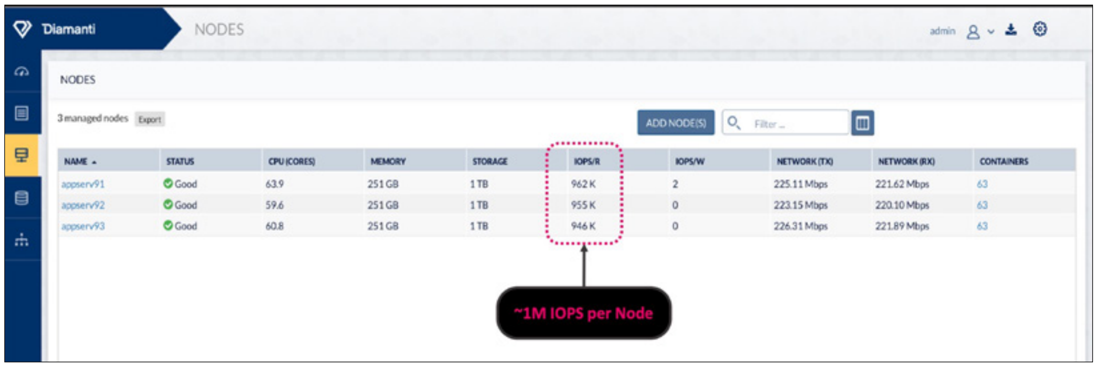


Figure 9: Load on the Diamanti cluster running pgbench (select-only query workload) with 63 PostgreSQL instances per node. Note complete CPU and memory saturation.

With the medium QoS tier, average IOPS was 15K per pod. With 63 PostgreSQL instances per node, that translates to approximately 945k IOPS per node. or 2.8M IOPS for the 3-node cluster. This is evident in the screenshot shown in Figure 10.



3 managed nodes Export

NAME	STATUS	CPU (CORES)	MEMORY	STORAGE	IOPS/R	IOPS/W	NETWORK (TX)	NETWORK (RX)	CONTAINERS
appserv91	Good	63.9	251 GB	1 TB	962 K	2	225.11 Mbps	221.62 Mbps	63
appserv92	Good	59.6	251 GB	1 TB	955 K	0	223.15 Mbps	220.10 Mbps	63
appserv93	Good	60.8	251 GB	1 TB	946 K	0	226.31 Mbps	221.89 Mbps	63

Figure 10: Load on each Diamanti node while running pgbench (select-only query workload) with 63 PostgreSQL instances per node. Note that each node delivers approximately 950k IOPS.

The IOPS numbers above correspond to a 4K transaction size, but because pgbench uses a mix of transaction sizes, the TPS numbers from pgbench don't match exactly. TPS numbers from pgbench are:

```
transaction type: SELECT only
scaling factor: 600
query mode: simple
number of clients: 16
number of threads: 2
duration: 300 s
number of transactions actually processed: 974827
latency average: 4.924 ms
tps = 3248.736280 (including connections establishing)
tps = 3249.048864 (excluding connections establishing)
```

In other words, pgbench reports 3.2k TPS per pod or about 200k per node with latency under 5 ms.

~200K TPS, >900k IOPS @ <5ms latency per node

It is important thing to note here that these are ***worst case numbers*** from a Diamanti platform perspective. We have used only 3.9 GB RAM for each PostgresSQL instance so that it will fault on index searches multiple times, which will result in multiple disk I/Os to satisfy a single select query. That is the reason each node is doing around 200k TPS but at the same time generating >945k IOPS to backend storage.

Looking at system-level metrics for CPU utilization using the Linux utility **top**, it is evident that the wait time for database instances is very low (2.7% wait; 98% CPU utilization). That means, even while generating over 945K IOPS, the CPU spent only 2.7% waiting for I/O, leading to a 98% CPU utilization for running queries.

```
top - 13:55:35 up 4 days,  2:34,  2 users,  load average: 866.20, 373.16, 142.85
Tasks: 3712 total, 794 running, 2918 sleeping,   0 stopped,   0 zombie
%Cpu(s): 67.4 us, 23.7 sy,  0.0 ni,  2.0 id,  2.7 wa,  0.0 hi,  4.3 si,  0.0 st
KiB Mem : 26385728+total,   640800 free,  9769048 used, 25344744+buff/cache
KiB Swap: 67108860 total, 66295812 free,   813048 used. 23976622+avail Mem
```

Even if the system was capable of more IOPS, it would not increase throughput further because we would run out of CPU resources before IOPS or network bandwidth.

Performance Results: TPC-B

We repeated the same test case using a write-heavy TPC-B like workload with **pgbench**. The outputs shown in this section are the same as those shown in the previous section. As above, CPU and memory utilization have been completely saturated.

```
TPC-B like transactions
0.085685   BEGIN;
0.679286   UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
0.180962   SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
0.184220   UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
0.183566   UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
0.153200   INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid,
:delta, CURRENT_TIMESTAMP);
0.345339        END;
```

Figure 11 shows consistent performance for all database instances running on Diamanti based on the IOPS/bandwidth curve.

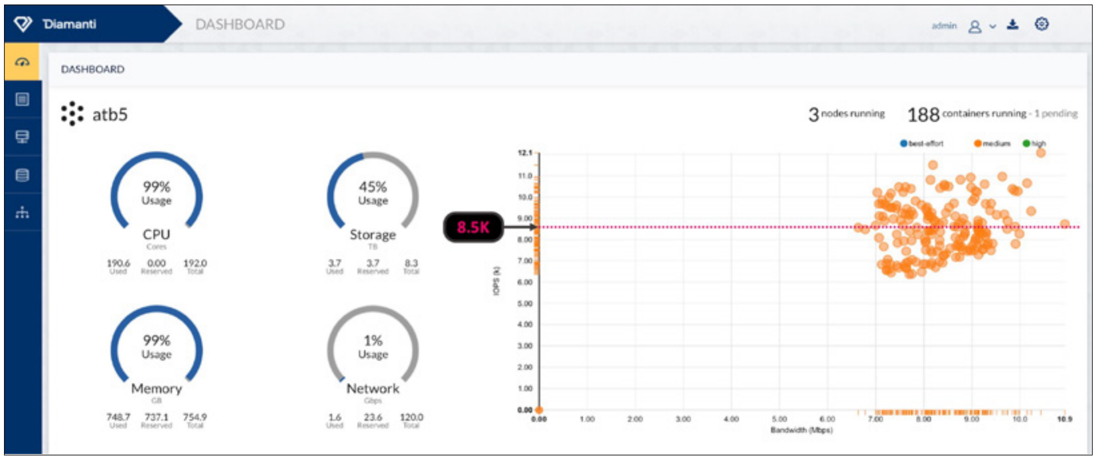
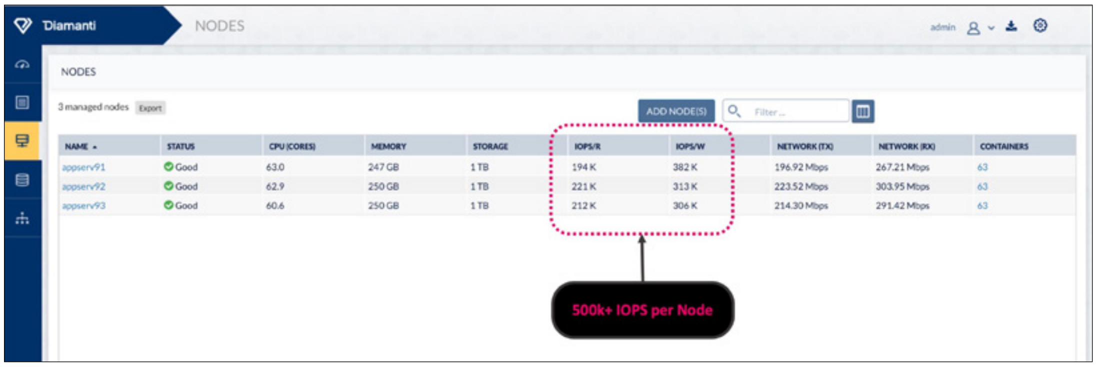


Figure 11: Load on the Diamanti cluster running pgbench (TPC-B like workload) with 63 PostgreSQL instances per node. Note CPU and memory saturation.

The IOPS seen on the Diamanti platform are shown on Figure 12.



NAME	STATUS	CPU (CORES)	MEMORY	STORAGE	IOPS/R	IOPS/W	NETWORK (TX)	NETWORK (RX)	CONTAINERS
appserv91	✓ Good	63.0	247 GB	1 TB	194 K	382 K	196.92 Mbps	267.21 Mbps	63
appserv92	✓ Good	62.9	250 GB	1 TB	221 K	313 K	223.52 Mbps	303.95 Mbps	63
appserv93	✓ Good	60.6	250 GB	1 TB	212 K	306 K	214.30 Mbps	291.42 Mbps	63

Figure 12: Load on each Diamanti node while running pgbench (TPC-B like workload) with 63 PostgreSQL i nstances per node. Note the write-heavy workload.

And the TPS results reported from pgbench are:

```
transaction type: TPC-B (sort of)
scaling factor: 600
query mode: simple
number of clients: 16
number of threads: 2
duration: 300 s
number of transactions actually processed: 180388
latency average: 26.609 ms
tps = 601.061482 (including connections establishing)
tps = 601.128876 (excluding connections establishing)
```

~37.8K TPS @ 26MS LATENCY PER NODE

Performance Results: Guaranteed Performance for Multi-tenant Environments

Next, we extended the testing to see if we could support different SLAs across various instances running on the same Diamanti cluster, as in a multi-tenant environment. By default, Diamanti supports three performance tiers for storage and networking. Additional customized performance tiers can be added as needed.

```
$ dctl perf-tier list
NAME           STORAGE IOPS   NETWORK BANDWIDTH   LABELS
best-effort    0              0                   <none>
high           15k            500M                <none>
medium         5k             125M                <none>
```

We distributed the workload according to three SLAs corresponding to these performance tiers:

- 20 **high**: 20×15k = 300k IOPS
- 20 **medium**: 20×5k = 100k IOPS
- 23 **best-effort**: not counted

TOTAL PROVISIONED IOPS = 400k

We then re-ran the same tests as above. For the select-only query workload we saw:

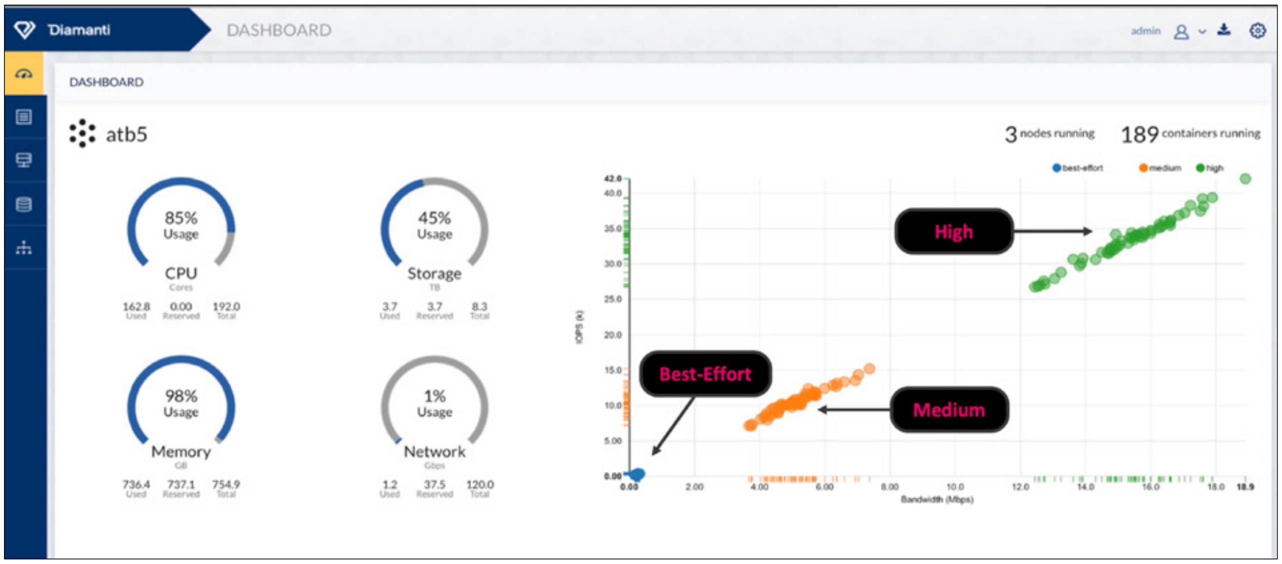


Figure 13: Load on each Diamanti node while running pgbench (select-only query workload) with 63 PostgreSQL instances per node: 20 high-performance instances, 20 medium-performance, and 23 best-effort.

And for the TPC-B like workload:

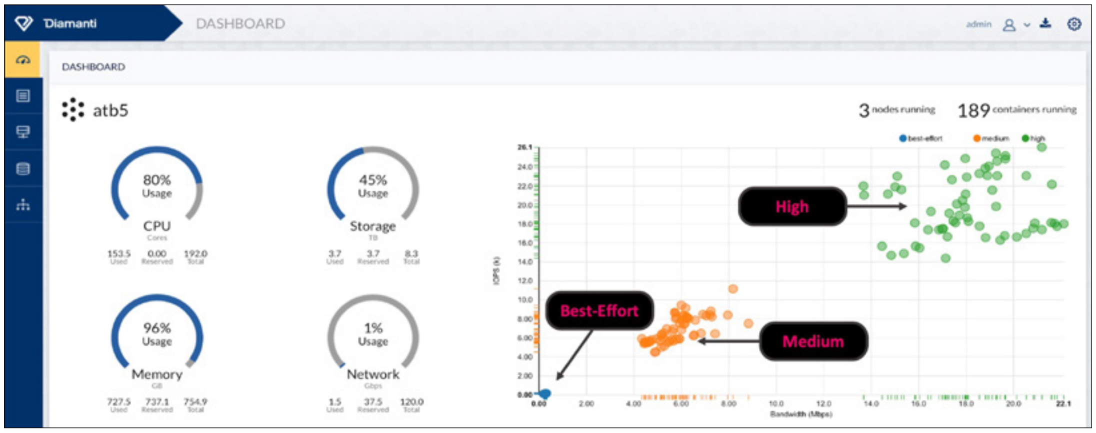


Figure 14: Load on each Diamanti node while running pgbench (TPC-B like workload) with 63 PostgreSQL instances per node: 20 high-performance instances, 20 medium-performance, and 23 best-effort.

This demonstrates that using Diamanti QoS capabilities you can adjust the relative priorities of various instances or tenants to meet specific service levels.

Table 2 and Figure 15 show results demonstrating how QoS improves the performance for critical jobs. With only 1 high tenant + 20 medium + 23 best-effort, you can achieve a *TPS of 15k @ 1 ms latency for select-query-only workloads AND TPS of 7.4k @ 2.1 ms latency for TPC-B like workloads.*

Table 2

Query type		Without QoS	With QoS (multiple high)			With QoS (critical load on high)		
		(63 per node)	High (20 per node)	Medium (20 per node)	Best-effort (23 per node)	High (1 per node)	Medium (20 per node)	best-effort (23 per node)
Select Only	TPS	3.2k	6.9k	2.7k	88	15k	2.9k	89
	Latency	5 ms	2.3	5.7	180 ms	1 ms	5.3 ms	177 ms
TPC-B	PS	600	1.5k	1.3k	14	7.4k	1.5k	15
	Latency	26 ms	10 ms	12 ms	1119 ms	2.1 ms	11 ms	1108 ms

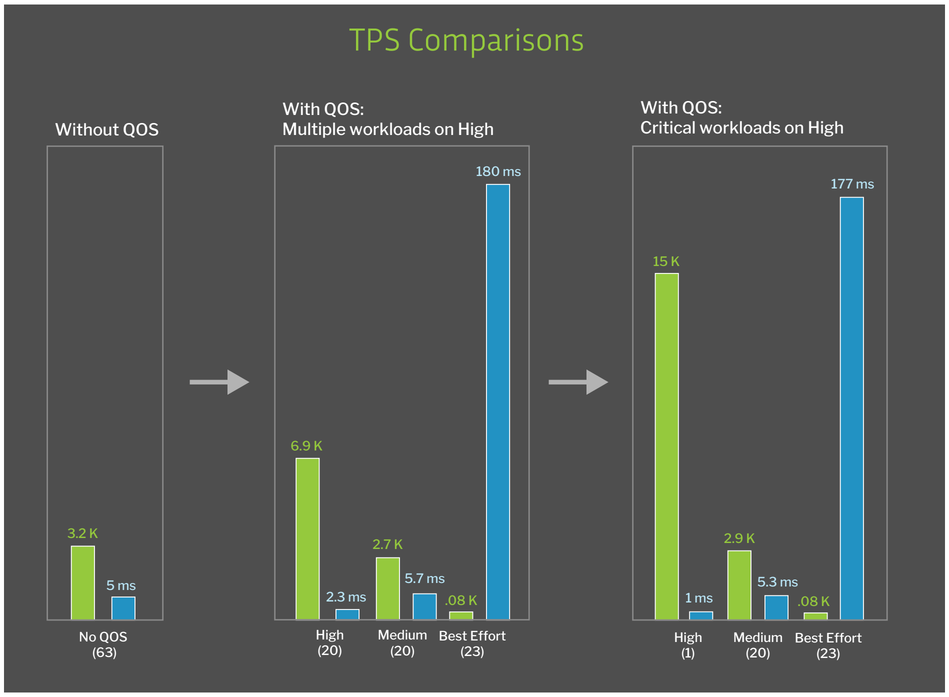


Figure 15: Designating a single instance as high-performance can deliver maximum throughput for a critical workload.

Achieving greater performance is just a matter of adjusting QoS. If *high* were to be defined as a minimum of 100K IOPS (as opposed to the default value of 15K IOPS), then TPS and latency numbers can be improved further, allowing you to optimize performance for your most important database workloads without sacrificing the performance of workloads in the **medium** tier.

Conclusion

Crunchy PostgreSQL for Kubernetes running on the Diamanti Enterprise Kubernetes Platform is a powerful solution for delivering "PostgreSQL-as-a-service". Crunchy Data's PostgreSQL Operator streamlines common database management operations including provisioning, scaling, backup/restore, and HA, making it much simpler for users to deploy and manage database services.

The Diamanti Enterprise Kubernetes Platform provides a turnkey solution for deploying modern, cloud native containerized workloads. It provides a bare-metal implementation, with PCIe level isolation for storage and network I/O resulting in a platform with the lowest TCO, especially for latency-sensitive workloads. Diamanti QoS features allow multiple tenants to share the same Kubernetes cluster without noisy neighbor problems, leading to very high overall platform utilization.

Acknowledgements:
Many thanks to Aditya Reddy and Steven Senecal for reviewing this report and providing valuable input and feedback.

About Crunchy Data:
Crunchy Data is the leading provider of trusted open-source PostgreSQL and enterprise PostgreSQL technology, support and training. Crunchy Data offers Crunchy Certified PostgreSQL, the most advanced true open-source RDBMS on the market. Crunchy Data is a leading provider of cloud native PostgreSQL – providing open-source, cloud-agnostic PostgreSQL-as-a-Service solutions. PostgreSQL's active development community, proven architecture, and reputation for reliability, data integrity, and ease of use makes it a prime candidate for enterprises looking for a robust relational database alternative to expensive proprietary database technologies. Learn more at www.crunchydata.com

About Diamanti:
Diamanti delivers the industry's only purpose-built, fully integrated Kubernetes platform, spanning on-premises and public cloud environments. We give infrastructure architects, IT operations, and application owners the performance, simplicity, security, and enterprise features they need to get cloud-native applications to market fast. Diamanti provides the lowest total cost of ownership to enterprise customers for their most demanding applications. Based in San Jose, California, Diamanti is backed by venture investors CRV, DFJ, Goldman Sachs, GSR Ventures, Northgate Capital, and Translink Capital. For more information visit www.diamanti.com or follow @DiamantiCom.