RUNNING DATABASES IN A KUBERNETES CLUSTER

An evaluation

Olle Larsson
Abstract

A recent trend in software engineering is to build applications composed of a set of small independent services – microservices. Kubernetes has become the common denominator for hosting stateless microservices. It offers foundational features such as deployment and replication of microservices as well as cluster resource management. Whereas stateless microservices are well suited to being hosted in Kubernetes, stateful microservices such as databases are generally hosted outside of Kubernetes and managed by domain experts. It is desirable to run stateful services such as databases in Kubernetes to leverage its features, ease of operation, and to harmonize the environment across the entire application stack. The purpose of this thesis is to investigate and evaluate the current support for hosting stateful applications in the form of databases in Kubernetes, and to show how different databases are able to operate in Kubernetes. An experimental setup was used where a set of databases – MySQL, TiDB, and CockroachDB, were deployed in a Kubernetes cluster. For each of these databases, a set of operational tasks were performed that concerned backup, upgrading, and capacity re-scaling. During the operations a number of server-sided and client-sided metrics related to the performance and resource efficiency of the databases were captured. The results showed that Kubernetes has got the native capabilities necessary to deploy and run databases, but not to fully operate them correctly. Furthermore, it was concluded that the operations had a widely different performance impact depending on the database solution.
Acknowledgements

Firstly, I would like to thank Elasitsys for providing me with this thesis. Secondly, I want to give a big thanks to my supervisor Johan Tordsson for guiding and providing me with valuable knowledge and feedback. Finally, I would like to thank Christian Klein for his enthusiasm and valuable input regarding the thesis.
Contents

1 Introduction 1
   1.1 Thesis goals 1
   1.2 Outline 2

2 Background 3
   2.1 Containers 3
   2.2 Kubernetes 3
      2.2.1 Architecture 3
      2.2.2 Kubernetes concepts 5
      2.2.3 StatefulSet 6
      2.2.4 Operators 6
   2.3 Databases 7
      2.3.1 MySQL 7
      2.3.2 TiDB 8
      2.3.3 CockroachDB 10
      2.3.4 Running CockroachDB in Kubernetes 10
   2.4 Metrics 10
   2.5 SysBench 11
   2.6 Database operations evaluated 11

3 Experimental setup 13
   3.1 Overview 13
   3.2 Capturing performance metrics 13
   3.3 MySQL 15
      3.3.1 Operations 15
   3.4 TiDB 16
      3.4.1 Operations 17
   3.5 CockroachDB 18
3.5.1 Operations

4 Results
4.1 Scale out
4.2 Scale in
4.3 Scale up
4.4 Scale down
4.5 Version upgrade
4.6 Backup
4.7 Summary

5 Conclusion
5.1 Future work

References

A Miscellaneous
A.1 SysBench configuration
A.1.1 MySQL
A.1.2 TiDB
A.1.3 CockroachDB
A.1.4 Table definition
A.2 cAdvisor
A.3 MySQL error
Chapter 1

Introduction

A recent trend in software engineering is to build applications following the microservice architecture pattern to facilitate scalability, ease of deployment and operation. The microservice architecture is an approach in which an application is built from a set of small, independent services - microservices [1]. Individual microservices are commonly delivered as so called containers. Containers are lightweight packages of software that contains everything it needs to run, such as the code, libraries, and dependencies [2]. Containers can run on different systems and they provide a predictable environment that is separated from other applications [3]. Most microservice are stateless, that is to say, they do not store any data. Consequently, the state, or the data, is usually stored in stateful storage such as a databases. This statelessness property simplifies microservice operations, they can easily be replicated and stopped without affecting the performance and the overall functionality of the application [4].

Kubernetes has become a popular platform for automated hosting of containerized microservices, offering foundational services such as deployment, scheduling, replication, and resource management. Kubernetes was originally designed to accommodate stateless microservices [5]. This means that while stateless microservices are being hosted in Kubernetes, stateful microservices are generally being hosted outside of Kubernetes and managed manually by domain experts such as database administrators. Kubernetes has started to improve its support for stateful microservices [6]. In the long run it is desirable to be able to host stateful microservices such as databases in Kubernetes to leverages its features, ease of operations, and to harmonize the environment across the entire application stack.

1.1 Thesis goals

The overall purpose of the thesis is to investigate and evaluate the current situation of hosting stateful microservices in the form of databases in Kubernetes. Databases are applications that are stateful in multiple different ways, they require persistent storage, they need to be uniquely identifiable, and they often have specific roles such as "master" or "slave" in larger database clusters. Furthermore, database client connections also has state and should not be broken unless necessary [7]. Therefore, databases should be a good indicator for the level of stateful support in Kubernetes.

The thesis aims to answer the following questions:

- How large is the gap between Kubernetes capabilities and the requirements of stateful
database services?
• How does different database solutions behave while being hosted in Kubernetes and operated on, and what is the impact of operating on both client performance and resource usage of the database?

The questions are answered by conducting a set of experiments. The experimental setup consists of deploying a few different database solutions using suitable configurations in Kubernetes and performing some operational tasks such as backup, upgrades, and capacity rescaling. While operating, a set of both server- and client sided metrics are recorded that relate both to performance and resource efficiency. These metrics forms the basis for answering both of the posed questions.

1.2 Outline

In Chapter 1 the topic of the thesis is presented along with the goals which the thesis aims to answer. Chapter 2 gives background on Kubernetes and the databases used for the experiments. In Chapter 3 the experiments along with the experimental setup is described. In Chapter 4 the results from running the experiments are presented. Finally, in Chapter 5 the thesis is concluded by discussing the questions in context of the experiments, and some potential future work is presented.
Chapter 2

Background

This chapter introduces the key technologies used in the thesis, including containers, Kubernetes and the evaluated databases. Some additional tools that was used for the experimental setup is also introduced and described.

2.1 Containers

A container is a piece of software that encapsulates code and all of its dependencies. This allows containers to be easily and consistently run in different environments. Containers provides benefits such as resource isolation, and they enable developers to focus on the software development and not on the environment in which the software is to be run [3].

Docker has become synonymous with containers and it is the most popular container format [8]. A docker image (image) is a template containing everything needed for an application to run as a container. It describes the constituting parts of the containers such as the code, files, and dependencies. A container is a running instance of an image [9].

2.2 Kubernetes

Kubernetes is an open-source portable platform for running and coordinating containerized workloads across multiple machines in a cluster. Kubernetes is designed to automate deployment, scaling, and management of containerized workloads throughout their entire life cycle [10]. Kubernetes, at a high-level, works by letting the user define the desired state of the cluster - what workloads there should be, how they interact with each other and the outside world, and Kubernetes ensures that the current state of the cluster is steered towards the desired state [11].

2.2.1 Architecture

The central part of Kubernetes is the cluster which is made up of nodes. A node is either a psychical machine or a virtual machine running Kubernetes. There are two types of nodes - master nodes and worker nodes. Master nodes and worker nodes serves different functions within a cluster. Usually master nodes are just referred to as masters, and worker nodes are referred to as nodes. Figure 1 shows the general outline for the architecture for Kubernetes [12].
The master node is the control plane for the cluster from which users and clients interact with the cluster to deploy and manage workloads. A cluster may have more than one master to achieve high-availability. A master node runs several different components. Following is a short presentation of each component:

- **etcd** - The cluster’s state and configuration is maintained in the etcd data store. etcd is a distributed key-value store and it is accessible by all nodes in the cluster.

- **kube-apiserver** - Through the kube-apiserver the master communicates with the rest of the cluster. It is also the main access point for users to access the control plane. A common way to interact with the kube-apiserver is through the command line client kubectl.

- **kube-scheduler** - The kube-scheduler is the component that is responsible for scheduling workloads to nodes. It keeps track of the capacity and resource usage of each node in the cluster and assigns workloads to nodes based on the collected information.

- **kube-controller-manager** - The kube-controller-manager manages controllers. Controllers are powerful constructs that runs control loops that through the use of the kube-apiserver manages the state of the cluster. Different controllers are responsible for different resources or parts of the cluster. Some controllers are responsible for managing the workloads in the cluster, while another is responsible for the nodes in the cluster.

- **cloud-controller-manager** - The cloud-controller-manager runs controller loops that are cloud-provider-specific, enabling Kubernetes to be run in the public clouds as well as on-premise. This also allows Kubernetes to abstract underlying implementations such as storage [11][12].

Worker nodes are responsible for running workloads. To be able communicate with the master and actually run workloads, a node needs to have the following components:

- **kubelet** - The kubelet is an agent that is responsible for managing the state of the node by making sure that containers are running and healthy. It collects performance information from the node itself, the containers, and shares it with the control plane through the kube-apiserver.
• **kube-proxy** - The kube-proxy is a network proxy for handling and maintaining network rules and forwarding network packets to the right container.

• **Container Runtime** - The container runtime is the software that allows containers to run. It is responsible for starting and managing containers as they are deployed to a node [11] [12].

2.2.2 **Kubernetes concepts**

In this section, some core Kubernetes concepts such as pods and ReplicaSets are introduced. Kubernetes objects are the entities in Kubernetes that represent the desired state of the cluster. Creating and manipulating Kubernetes objects is done through the Kubernetes API that is exposed by the Kubernetes master. To create a Kubernetes object one has to first provide a description of the desired state for the object. This description is usually expressed in a YAML Ain't Markup Language (YAML) file. When the object description is submitted to the Kubernetes API, Kubernetes creates the object to make sure that the desired state of the cluster is met [13].

A pod is the smallest deployable unit of work and the most basic object in Kubernetes. Figure 2 shows the general outline for a pod. As can be seen, a pod consists of one or more containers. Even though pods can consist of multiple containers, it is most common to just run one container per pod. Each pod has a unique IP address assigned to it, and containers within a pod can communicate with each other through localhost. Pods can have volumes attached for storage [14]. A commonly used mechanism are so-called *Init Containers*, which are containers that run to completion before the main application containers are started. They can be used to perform tasks such as to setup and pre-populate configuration files [15]. In multi-container pods a *sidecar* is a utility container that runs alongside the main application container with the purpose to enhance the main container by providing extra functionalities such as monitoring, advanced networking, etc [16].

Pods are rarely created directly because they are designed to be ephemeral objects. If a created pod gets killed, for example, then a user would have to manually create another pod. Even though pods can be created directly they are often created and managed by higher-level abstractions [14].

The ReplicaSet is an higher-level abstraction that makes sure that a specified number of replicas of a pod are always up and running. If one replica goes down, it creates another one, thus making sure that desired number is met. The ReplicaSet enables users to increase or decrease the number of replicas when needed. While the ReplicaSet enables scaling of pods, it does not offer any other management capability such as updates [17]. Therefore, the ReplicaSet is seldomly deployed and managed by users. Instead higher-level abstractions such as the Deployment is used that leverages the ReplicaSet. The Deployment allows for both scaling and updating of pods. In a Deployment pods are created in a random order, they are assigned a random id, and they are treated symmetrically [18]. The properties of the Deployment are not suitable for running databases, or more specifically databases clusters that consist of multiple database nodes. In a database cluster nodes are not symmetric, they have specific roles such as master or slave and they need to be uniquely identifiable and discoverable across restarts. While the
Kubernetes Deployment does not provide and satisfy the needs for running databases, the StatefulSet does.

2.2.3 StatefulSet

The StatefulSet, previously known as PetSet, is a higher-level abstraction that is built for (replicated) stateful applications such as databases. The basis of the StatefulSet is that it manages the deployment, scaling, and updating of a set of replica pods. The StatefulSet provides a set of guarantees regarding the ordering and uniqueness of pods and pod resources. In a StatefulSet each pod has a unique, persistent identity and hostname that is maintained in the event of restarts. Each pod also has stable persistent storage attached to it.

The StatefulSet uses an ordinal index to provide a deterministic order of pod creation and termination. Pods are created in sequential order and they are terminated in reverse ordinal order. The ordinal index is used to provide uniqueness to the pod identity. A pod’s identity consists of the name of the StatefulSet and the pod’s ordinal index, e.g., db-0, where db is the name of the StatefulSet and 0 is the ordinal index.

The guarantees of ordered initialization of the pods starting with ordinal index 0 is useful for applications in which there are different roles such as master and slave in certain databases. The ordered initialization enables pods to be configured differently depending on the ordinal index. For example, in a database cluster it is common to deploy the master before the slaves, and thus, the master node is initialized first as the pod with ordinal index 0. When the slaves are initialized they can be configured as slaves, and they know that the master is reachable at the pod with ordinal index 0.

By default, when updating a StatefulSet, such as changing the container image, the update strategy RollingUpdate is used. The update strategy tells the controller how it should manage the pods when updating. The update strategy RollingUpdate is an automatic process where the controller in reverse ordinal order, terminates and recreates pods one at a time.

2.2.4 Operators

Kubernetes StatefulSet is a big building block for enabling databases to be run in a Kubernetes cluster. However, databases are complex stateful applications that usually require some application-specific knowledge to be managed correctly. To this end, a StatefulSet cannot decode any application-specific knowledge of the application that is running inside its pods.

The concept of operators was introduced by CoreOS. They describe an operator as a piece of software that implements application specific knowledge to create, configure, and manage complex stateful applications in Kubernetes. An operator encapsulates the operational logic required to automate some common operational tasks that a user would have to handle manually otherwise.

Operators extends the Kubernetes API by creating custom resources and controllers. The custom controllers leverages the basic Kubernetes primitives such as the StatefulSet, and implements the application-specific knowledge. Two well known operators are the etcd Operator and the Prometheus Operator.

As such, the operator framework seems like a promising approach to implement complex management operations for databases, enabling these to be run in Kubernetes.
2.3 Databases

In this section the databases used for the experiments are presented. The databases are MySQL, TiDB and CockroachDB. MySQL is a database that most software developers either have experience working with or have at least heard of, whereas both TiDB and CockroachDB are fairly new databases and far less common. The databases have a couple of features that they all have in common. They all use the relational data model in which data is modeled as rows and columns in tables. Furthermore, they all use the Structured Query Language (SQL) to query and manipulate data, and they all support transactions with full ACID (Atomicity, Consistency, Isolation, Durability) semantics. In the following sections each database solution is introduced in little bit more detail.

2.3.1 MySQL

MySQL is the world’s most popular open-source relational database [23]. It used by many companies such as Google, Facebook, and YouTube. There exists some variants of the database but the core is the MySQL server [24]. MySQL uses SQL to query and manipulate data. The database supports transactions with full ACID semantics. Even though MySQL is a relational database it can be used as a document store as well [25].

Scaling a MySQL database horizontally can be done using a couple of different methods. The first method is called sharding which involves partitioning the database into smaller chunks (shards) and distributing them across a cluster of servers. The second method that can be used is called replication. There exists many different types of replication. One common type is called master-slave replication. In master-slave replication one server acts as the master and the rest of the servers acts as slaves. All queries that modify the database are handled by the master. The database modifications are later propagated from the master to the slaves. Read queries can also be handled by the slave servers, thus increasing the read performance of the database cluster [26]. Figure 3 shows a simple master-slave replication setup consisting of one master and one slave.

2.3.1.1 Running MySQL in Kubernetes

Perhaps the most promising approach to run MySQL in Kubernetes is to use Presslabs MySQL Operator. The operator is still in alpha stage, and is therefore most likely not suitable for any sort of mission-critical production workloads.

The operator deploys and manages highly-available MySQL clusters that are based on the Percona server for MySQL 5.7. The clusters uses asynchronous master-slave replication and the operator leverages Github’s Orchestrator. Orchestrator is a MySQL tool for managing replica-

Figure 3: A simple illustration of a master-slave replication setup. Write queries are directed to the master and the read queries can be handled by slaves as well.
tion topologies and providing high availability [27]. Each database node comes with a built-in metrics exporter that makes it easy to integrate with monitoring systems. The operator supports recurrent backups. The backups can only be stored on object storage services. Currently these storage services include Google’s Cloud Storage (GCS), Amazon’s S3, and Azure’s Blob Storage [28]. Clusters can also be restored from backups [29].

Figure 4 shows a high-level overview of what a deployment of the operator and a MySQL cluster looks like. As can be seen the operator deploys two pods: the operator itself and Github’s Orchestrator. A MySQL cluster is implemented as a StatefulSet with a set of replicated pods. Each pod runs a few containers as can be seen in the figure. The blue-colored containers are the Init containers which are used when a pod is being initialized. They set up the configurations file for the Percona MySQL server and clones or restores the database if there is one. The green-colored containers are the containers that run indefinitely. The main container of the pod is the Percona MySQL server and the others are sidecar containers that provides utility and functionality to the main container. The sidecar container provides functionality such as backup, pt-heartbeat provides the necessary information to orchestrator, and the metrics-exporter exports MySQL related metrics from the MySQL server that can be collected by monitoring solutions.

2.3.2 TiDB

The “Ti” in TiDB is an abbreviation for Titanium, however, the database is named TiDB. TiDB is an open-source distributed SQL (NewSQL) database that is built on top of a distributed transactional key-value store (TiKV). TiDB is developed by the Chinese company PingCAP, and it supports both analytical and transactional workloads. TiDB supports the MySQL wire protocol and most of its syntax, and can therefore be used as a drop-in replacement for MySQL. Before using TiDB as a replacement for MySQL it is advised to investigate which features are unsupported by TiDB [30]. Some of TiDBs main features are that it scales horizontally, supports distributed transactions with strong consistency, and high availability.

The architecture for TiDB can be seen in Figure 5. The three main components constituting TiDB are: the TiDB server, the TiKV server, and the and Placement Driver (PD) server [31].
As shown in Figure 5, the TiDB server is the component of the database that is responsible for handling client connections and SQL requests. An application sees a TiDB server as a MySQL 5.7 server. The TiDB server does not store any data, thus making it stateless. Because of the statelessness it easy to scale the TiDB server cluster horizontally.

The TiKV server is a distributed transactional key-value store. It is the component responsible for storing data. The basic unit of data is called a region and it is a continuous chunk of the key-value space. By default whenever a region grows to 96 MB, it is split in two. Each region is also by default replicated three times to ensure consistency and fault tolerance. Replication is done using the Raft consensus algorithm. Each range and its replicas that are distributed across different nodes constitutes a Raft group.

The Placement Driver (PD) manages and coordinates the entire cluster. The PD server, among other things, ensures that each range is sufficiently replicated and that ranges are evenly distributed across the TiKV cluster. It is also responsible for storing other metadata about the cluster required by both TiDB and TiKV. The PD server also uses the Raft consensus algorithm to ensure redundancy and high availability.

2.3.2.1 Running TiDB in Kubernetes

The most promising approach to run TiDB in Kubernetes is to use the TiDB-Operator. The operator is developed by PingCAP and it is an operator for deploying and managing TiDB clusters. The operator automates operational tasks such as scaling and updating TiDB clusters. The operator is still in beta and might therefore not be suitable for production environments.

The TiDB operator supports both incremental backups and full backups. The full backups can be scheduled to be run periodically. Backups are by default stored on disk but they can also be stored in either GCS or in a Ceph bucket. The operator uses a program called mydumper to provide some of the backup and restore functionality. The TiDB operator creates

---

Figure 5: Architecture for the TiDB database. Three main components: TiDB server, TiKV server, and PD server.
and configures a monitoring solution for monitoring the performance of the TiDB cluster. 

2.3.3 CockroachDB

CockroachDB is an open-source NewSQL database that is built on-top of a distributed transactional key-value store. CockroachDB is developed by Cockroach Labs, and it supports transactional workloads. CockroachDB is designed to be horizontally scalable, consistent, highly available, and failure tolerant. CockroachDB supports the PostgreSQL wire protocol and might therefore be used as a drop-in replacement for PostgreSQL.

A CockroachDB cluster consists of one or more CockroachDB nodes. All nodes are symmetric in the sense that no node has any special role within a cluster and each node can serve clients requests. This symmetric property also facilitates container-based deployments. CockroachDB is deployed as a single self-contained binary.

The smallest unit of data is called a range. A range is a continuous chunk of the key-value space. When a range grows to 64 MB it is split in two. By default, ranges are replicated three times through the use of the Raft consensus algorithm. For each range, one of the replicas has the role of the Leaseholder. The leaseholder is the replica which manages and coordinates all read and write requests for that range. Usually, the leaseholder replica is the Raft group leader for that range as well.

2.3.4 Running CockroachDB in Kubernetes

There exists one Kubernetes operator for CockroachDB and it is made by Rook. The operator deploys CockroachDB clusters, but what features the operator provides is not known due to the lackluster documentation. After some brief initial testing of using the operator it was concluded that the operator provides no further operational features then what an ordinary Kubernetes StatefulSet does. Therefore, the best approach, as of now, to run CockroachDB in Kubernetes is to deploy it using a StatefulSet.

2.4 Metrics

This section introduces and describes the metrics that are used to determine the impact of operating on the databases in Kubernetes. The metrics are divided into two categories, one that contains the client-sided metrics that relates to the client performance, and one for the server-sided metrics that relates to the resource usage for the databases.

Following is a short description for the server-sided metrics.

- **CPU usage** - Is the total CPU usage for the database.
- **Memory usage** - Is the total memory, measured in bytes, consumed by the database.
- **Network receive rate** - Is the number of network bytes received by the database.
- **Network transmit rate** - Is the number of network bytes transmitted by the database.

Following is a short description of the client-sided metrics.

- **Throughput** - Is the number of transactions the server can respond per second. Is measured in transactions per second (TPS).
- **Latency** - Is the delay, or the user perceived response time, measured in milliseconds. Essentially, it is the time from the issue of a transaction until the response has been returned from the database.

### 2.5 SysBench

SysBench is a synthetic benchmarking tool for benchmarking databases and the hardware that they run on. SysBench was originally designed to only work with MySQL but it now supports PostgreSQL. The program is written in C and executes benchmarks that consists of workloads written in the scripting language **LUA**. LUA is considered to be one of the fastest scripting languages and it is designed to be embedded into C or C++ applications. The LUA scripts implements pre-defined hooks that are called from C code. This makes SysBench very versatile since one can, for example, define a custom workload for a highly specific use case. SysBench comes with a set of scrips for benchmarking OLTP performance of SQL databases. OLTP stands for Online Transaction Processing and it refers to transactional workloads such as those for online systems such as e-commerce or systems in the financial sector.

SysBench outputs intermediate reports at a user configurable interval. The intermediate reports contains metrics such as transactions per second, queries per second, and latency. The format for the intermediate reports can be customized by providing a new implementation for that hook. When a benchmark has been completed SysBench displays a cumulative report for the entire run.

SysBench has the ability to run infinite benchmarks, which makes it suitable as a workload generating tool as well as a benchmarking tool [39].

### 2.6 Database operations evaluated

This section introduces and describes the operational tasks, or just operations, that were used in the experiments for operating the databases. The operations concerned scaling, upgrading, and backing up data. More specifically the following operations were used: *scale up, scale down, scale out, scale in, version upgrade*, and *backup*.

The *backup* and *version upgrade* are operations that are quite self explanatory, backup is the operation where a copy of the data in the database is created and stored, and version upgrade is the operation where the database software that is running is upgraded.

*Scale up* and *scale down* are scaling operations that are also known as *vertical scaling*. Figure 7 shows the concept of scaling out and scaling in. As can be seen, scaling up is the process where more resources such as CPU and memory are added to the machine that the database is running on, while scaling down is the reverse process, where resources are removed from the machine.

*Scale out* and *scale in* are scaling operations that are also known as *horizontal scaling*. Figure 6 shows the concept of the operations. Scaling out is the operation where more nodes (database servers) are added to the system or cluster, and scaling in is the reverse operation where nodes are removed from the cluster [40].
Figure 6: Illustration of the concept of scaling up and down (vertical scaling).

Figure 7: Illustration of the concept of scaling out and in (horizontal scaling).
Chapter 3

Experimental setup

The experimental setup described in this chapter was used to gather the necessary information to answer the questions posed in Section 1.1. The experiments consisted of deploying each database solution in a suitable configuration on Kubernetes and performing a set of operational tasks concerning backup, scaling, and upgrading. While the operations were being executed a set of client- and server-side metrics were captured as to determine the impact of operating on both servers and clients.

3.1 Overview

The experiments were performed on a Kubernetes cluster created by the Google Kubernetes Engine (GKE). GKE is a managed Kubernetes service which means that the setup and management of the cluster is taken care of. GKE is a part of the Google Cloud Platform (GCP) and GKE can thus utilize features that GCP provides. One such feature is node pools. A node pool is a group of nodes that all share the same configuration. With node pools it is easy to have groups of machines in a cluster that, for example, have different hardware setups [41]. The GKE version 1.11.8-gke.6 was used. A data set was prepared by SysBench that consisted of 25 tables with 1 million rows in each table. The data set was approximately 6 GB in size [2]. The persistent storage for each database node consisted of a 100 GB, network attached, SSD persistent disk [3]. SysBench version 1.1.0 was used at the client side. It generated transactional workloads for the databases and captured throughput and latency as seen by the client.

Because of the innate differences between the databases used, the setup for every database was unique. In the following sections the setup used to captured the server-side metrics from the databases is first presented. Next presented is the experimental setup for each database.

3.2 Capturing performance metrics

Capturing the server side metrics from the database nodes or pods was done through the use of the components shown in Figure 8. As can be seen the setup consisted of three components:

---

1. See Appendix A.1.4 for the table definition.
2. When looking at the data size in MySQL it might vary slightly for both CockroachDB and TiDB.
3. For more info on the disks used see https://cloud.google.com/persistent-disk/
The first component in the monitoring stack is **cAdvisor** (Container Advisor). It is a daemon that collects and exports container metrics such as CPU, memory, and network usage. The cAdvisor collects these metrics at a fixed interval from all containers running on the host machine. What metrics to collect, and at which interval, can be configured [42]. The cAdvisor is a part of the kubelet component that runs on each Kubernetes node. However, it is not always possible to configure the cAdvisor that is deployed as a part of the kubelet. Such is the case when using GKE and therefore cAdvisor was deployed as a standalone component.

The cAdvisor was deployed using a Kubernetes DaemonSet. The DaemonSet guarantees that all Kubernetes nodes runs a replica of a pod [43]. This deployment model was optimal for cAdvisor as it had to be run on all hosts. See Appendix A.2 for the details regarding the configuration and deployment of cAdvisor.

The second component in the monitoring stack is **Prometheus**. Prometheus is an open-source monitoring and alerting system. Prometheus was originally built by SoundCloud but is now a part of CNCF. Prometheus collects metrics by pulling or scraping targets or services (HTTP endpoints) that expose Prometheus formatted metrics. Prometheus stores the collected metrics in a time series database. The **Prometheus Query Language** (PromQL) is used to select and aggregate the time series data [44]. As shown in Figure 8, Prometheus scraped the container metrics collected by cAdvisor.

The final component in the monitoring stack is **Grafana**, an open-source monitoring and metrics visualization tool. Grafana supports querying multiple different data sources, including Prometheus. As can be seen in Figure 8, the Grafana instance queried Prometheus using the PromQL to select and aggregate the time series data. The metrics were visualized in a set of graphs.

Both Prometheus and Grafana was deployed and configured using the **Prometheus Operator**. The Prometheus Operator is built by CoreOS and it creates, configures, and manages Prometheus instances in Kubernetes [45]. The operator enables configuration of which services Prometheus should collect metrics from. This is all done through the use of Kubernetes primitives, which essentially abstracts the domain specific knowledge required to configure Prometheus away from the user [46].
3.3 MySQL

The for Presslabs MySQL Operator, version 0.2.7, was used to deploy and operate MySQL. Figure 9 shows the experimental setup that was used for MySQL. Initially the deployment consisted of a MySQL cluster with two nodes, one master and one slave. The MySQL nodes ran the Percona MySQL 5.7.24 image.

Two Sysbench instances were used to generate load on the master node. One instance issued transactions that only consisted of read queries and the other instance issued transactions that only consisted of write queries. The instance which issued the write-only transactions was limited to a rate of 40 TPS. For the slaves, a single SysBench instance was used that issued read-only transactions at an unlimited rate.

The reason for separating the read and write workload on the master node is due to the possibility of the operator performing a failover while operating, where it promotes a slave to be the master. To avoid the throughput being completely dropped on the old master in the event of such a failover, the read and write loads were separated so that the old master still can serve the read workload.

<table>
<thead>
<tr>
<th>Node pool</th>
<th>Size</th>
<th>Machine type</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monitoring</td>
<td>1</td>
<td>n1-standard-2</td>
<td>2 vCPU, 7.5 GB memory</td>
</tr>
<tr>
<td>Sysbench</td>
<td>1</td>
<td>n1-standard-2</td>
<td>2 vCPU, 7.5 GB memory</td>
</tr>
<tr>
<td>Small</td>
<td>3</td>
<td>n1-standard-4</td>
<td>4 vCPU, 15 GB memory</td>
</tr>
<tr>
<td>Big</td>
<td>2</td>
<td>n1-standard-8</td>
<td>8 vCPU, 30 GB memory</td>
</tr>
</tbody>
</table>

Table 1 shows the Kubernetes cluster used for the MySQL experiments. As can be seen the cluster consisted of 4 different node pools for a total of 7 Kubernetes nodes. The monitoring stack was deployed on the Kubernetes nodes pertained by the node pool Monitoring. The MySQL cluster was initially deployed on the Kubernetes nodes pertained by the node pool Small. The SysBench instances were deployed on the node in by the node pool Sysbench.

3.3.1 Operations

This section describes the implications on the database when the operations were performed.

- When scaling up, the database nodes were moved from the nodes that they were initially deployed on (Small), to the nodes pertained by the node pool Big. Conversely, when scaling down, the database nodes were moved back to Small nodes that they were running on before the scale up operation.

- For scale out, a second slave was added to the cluster. Conversely, for scale in the second slave was removed.

---

7 More specifically the following image was used: percona@sha256: c8b69b3c753cb04f1cbf8a4a1f2b95f51b5756e1ee6368ad7808a5205e2d45cfeb
8 See Appendix A.1.1 for the complete setting for SysBench
9 Given by the workload in: `oltp_read_only.lua`
10 Given by the workload in: `oltp_write_only.lua`
• When backup was performed, one of the nodes in the cluster was chosen to perform the backup by streaming the data to GCS.

• During the version upgrade operation, the nodes were upgraded to a newer version of the Percona MySQL image.

Figure 9: This figure illustrates a high-level overview of the experimental setup for the MySQL database. The monitoring stack is excluded along with the actual mapping of the database nodes on Kubernetes nodes. Initially the cluster consisted of two nodes, one master and one slave. Each node was backed by 100 GB of SSD persistent storage.

3.4 TiDB

The TiDB Operator, version 0.1.0, was used to deploy and manage TiDB. Figure 5 shows the experimental setup for TiDB. As can be seen the initial setup consisted of 2 TiDB servers, 3 PD servers, and 3 TiKV servers. Each TiDB server had a SysBench instance connected that issued read-write transactions at an unlimited rate.

11 More specifically the following image was upgraded to: percona@sha256: b3b7fb177b416563c46fe012298e042ec1607cc0539ce6014146380b0d27b08c
12 See Appendix A.1.2 for the complete configuration used for the SysBench instances.
13 Given by oltp_read_write.lua
Table 2: The constituting node pools that made up the Kubernetes cluster for the experimental setup for the TiDB database. The table describes the name, the size, the machine type and the hardware for each node pool.

<table>
<thead>
<tr>
<th>Node pool</th>
<th>Size</th>
<th>Machine type</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monitoring</td>
<td>1</td>
<td>n1-standard-2</td>
<td>2 vCPU, 7.5 GB memory</td>
</tr>
<tr>
<td>Sysbench</td>
<td>1</td>
<td>n1-standard-2</td>
<td>2 vCPU, 7.5 GB memory</td>
</tr>
<tr>
<td>TiDB-small</td>
<td>3</td>
<td>n1-standard-2</td>
<td>2 vCPU, 7.5 GB memory</td>
</tr>
<tr>
<td>TiDB-big</td>
<td>3</td>
<td>n1-standard-4</td>
<td>4 vCPU, 15 GB memory</td>
</tr>
<tr>
<td>TiKV-small</td>
<td>4</td>
<td>n1-standard-2</td>
<td>2 vCPU, 7.5 GB memory</td>
</tr>
<tr>
<td>TiKV-big</td>
<td>3</td>
<td>n1-standard-4</td>
<td>4 vCPU, 15 GB memory</td>
</tr>
</tbody>
</table>

Table 2 shows the Kubernetes cluster used for the TiDB experiments. As can be seen the cluster consisted of 6 node pools for a total of 15 Kubernetes nodes. The monitoring stack was deployed on the Kubernetes nodes pertained by the node pool Monitoring, and the SysBench instances was deployed on the node pertained by the node pool Sysbench. In the initial deployment of TiDB the TiDB server cluster and the PD server cluster were deployed in the TiDB-small node pool, and the TiKV server cluster was deployed in the node pool TiKV-small.

The TiDB servers and the PD servers were running on the same Kubernetes nodes, while the TiKV server were running on another set of nodes. Machine sizes were smaller for TiDB compared to MySQL (Table 2 vs. Table 1). However, the combined hardware resources of the nodes running a TiKV, PD and TiDB instance, matched the hardware used for a single MySQL server.

Figure 10: This picture illustrates a high-level overview of the experimental setup for TiDB. The monitoring stack along with the actual mapping of the database nodes on Kubernetes nodes are excluded. The TiKV nodes were backed by 100 GB SSD persistent storage, and the PD nodes were backed by 10 GB SSD persistent storage.

3.4.1 Operations

Due to the componential architecture of TiDB, the scaling operations were performed a bit differently compared to MySQL. The scale up and scale down operations were performed on each component, starting with the PD servers and ending with the TiKV servers. The scale out and scale in operations were performed on the TiDB server cluster and the TiKV server cluster, starting with the TiDB server cluster and ending with the TiKV server cluster. The
version upgrade operation was also performed for each component, starting with the PD servers and ending with the TiKV servers.

Following is a short description of the implications for each operation on the database.

- Upon *scale up*, the PD servers and the TiDB servers were moved from the nodes that they were initially deployed on to the nodes in the *TiDB-big* node pool. For *scale down*, they were moved back to the nodes that they were initially deployed on.
  Similarly, the TiKV servers were moved to and from the *TiKV-big* node pool upon scale up and scale down, respectively.

- When *scale out* was performed on the TiDB server cluster, another server was added, thus making the cluster consists of three servers. When *scale in* was performed, the third server was removed.
  In the same manner, a fourth TiKV server was added, and removed, upon scale out and scale in, respectively.

- Upon *backup*, the operator created another pod that collected the entire data set, which was then streamed to GCS.

- When *upgrading version*, the PD servers, TiDB servers, and the TiKV servers were all upgraded from version v2.1.5 to version v2.1.8.

### 3.5 CockroachDB

Figure 11 shows the experimental setup that was used for CockroachDB. The initial setup consisted of a CockroachDB cluster with three nodes. CockroachDB was deployed using a *StatefulSet*. Version v2.1.6 of CockroachDB was used. Each node had a SysBench instance connected to it. The SysBench instances had four client connections that issued read-write transactions at an unlimited rate.\(^{14}\)

<table>
<thead>
<tr>
<th>Node pool</th>
<th>Size</th>
<th>Machine type</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monitoring</td>
<td>1</td>
<td>n1-standard-2</td>
<td>2 vCPU, 7.5 GB memory</td>
</tr>
<tr>
<td>Sysbench</td>
<td>1</td>
<td>n1-standard-2</td>
<td>2 vCPU, 7.5 GB memory</td>
</tr>
<tr>
<td>Small</td>
<td>4</td>
<td>n1-standard-4</td>
<td>4 vCPU, 15 GB memory</td>
</tr>
<tr>
<td>Big</td>
<td>3</td>
<td>n1-standard-8</td>
<td>8 vCPU, 30 GB memory</td>
</tr>
</tbody>
</table>

Table 3 shows the Kubernetes cluster used for the CockroachDB experiments. As can be seen, the cluster consisted of 4 node pools for a total of 9 Kubernetes nodes. The monitoring stack was deployed on the node in the node pool *Monitoring*, and the Sysbench instances were deployed on the node in the node pool *Sysbench*. The CockroachDB cluster was deployed on the nodes in the node pool *Small*.

\(^{14}\)See Appendix A.1.3 for the complete settings for SysBench.
Figure 11: This figure illustrates a high-level overview of the experimental setup used for the CockroachDB database. The monitoring stack as well as the actual mapping of the database nodes to Kubernetes nodes are not included. Initially the cluster consisted of three nodes. Each node was backed by 100 GB of SSD persistent storage.

3.5.1 Operations

Operating the database was done by leveraging the StatefulSet features and native CockroachDB commands. The scaling and upgrading operations were executed through the use of the StatefulSet, while the backup operation had to be manually issued using a CockroachDB command. Following is a short description of the implications for each operation on the database.

- Upon **scale up**, the database cluster was moved from the nodes that they were initially deployed on, to the nodes pertained by the node pool *Big*. Conversely, when **scaling down**, the database cluster was moved to the nodes that they were running on prior to the scale up operation.

- When **scaling out**, a fourth node was added to the cluster. Conversely, when **scaling in**, the fourth node was removed from the cluster.

- For **backup**, the CockroachDB node with ordinal index 0 was issued to dump the data, which created a file that contained the SQL commands necessary to restore the database.

- Upon **version upgrade**, the database nodes were upgraded from version v2.1.6 to v2.1.7.
Chapter 4

Results

This section presents the results gathered from running the experiments. The results are presented operation-wise starting with scale out and ending with backup. For each database solution, the server side metrics are shown alongside the client side metrics. While memory was one of the server side metrics that were gathered, it is not shown due to the fact that it did not contribute any valuable information. To see the impact of operating, the metrics are shown some time prior to the start of the operation, which is highlighted with a vertical line in all plots, and labeled as time 0. The throughput presented is the aggregated throughput from all the client instances, and the latency presented is the average latency from all the client instances in the setup.

4.1 Scale out

Figure 12 shows the results from scaling out the different databases. As can be seen, the impact across the databases is quite different. Overall, network rates were the metrics impacted the most by the operation across the databases. The most notable result is the lack of impact on the client performance while scaling out the MySQL cluster, see Figure 12b. The figure shows that when scaling out the MySQL cluster, the new slave mysql-2, cloned the data of the first slave, mysql-1, at a rate of about 10 MBps.

Figure 12a shows the result from performing the operation on CockroachDB. Once the cluster was scaled out and the fourth node, cockroachdb-3, had joined the cluster, a rebalancing of cluster data to the new node was initialized. The data-rebalancing process was the reason for the increase of the network receive rate for the fourth node, and the increase of the transmit rates for the other nodes. It is worth to note that the rebalancing of the data almost took 35 minutes to complete. Once the new node joined the cluster, it started to accept client requests. As it had no data, the other nodes had to serve the requests, thus increasing the overall throughput but also the latency. As more and more data was rebalanced to the new node, the latency was decreased.

Figure 12c and 12d shows the results from scaling out the TiDB server cluster and TiKV server cluster respectively. As can be seen, when the TiDB server cluster was scaled out, the TiKV servers had their resource usage increased due to there being more load in the cluster. When the TiKV cluster was scaled out, there was a negative impact on the client metrics for about two minutes. Once the new node, tikv-3, had joined the cluster, as with CockroachDB, the data in the cluster had to be rebalanced to the new node. The data-rebalancing process
was the reason for the large increase of the network receive rate for the fourth node, and for
the large increase of the network transmit rates for the other nodes. It is worth noting that
the data-rebalancing process took about 5 minutes, which can be compared to 35 minutes for
CockroachDB.

It is worth noting that in order to even be able to even perform the operation on the MySQL,
the client instance that issued write transactions on the master node, mysql-0, had to be
limited to a rate of 40 TPS. When the rate was higher the operation was aborted due to an
error occurring in the sidecar container of the node from which the new node was cloning
data. The error essentially stated that it could not copy the data due to it being too slow. After
some investigation it was found that the sidecar container in a MySQL node is only allowed
to use 0.05 vCPU. This might be one of the reasons why the operation could not successfully
be performed when the write load was higher. This limit was likely also the reason for the
lack of impact on CPU usage when scaling out the MySQL cluster.

---

1See Appendix A.3 for error message.
Figure 12: Impact of scale out operation on client performance and server resource usage for CockroachDB (a), MySQL (b), and TiDB (c and d)

4.2 Scale in

Figure 13 shows the results from scaling in the different database clusters. As can be seen, the operation impacts the databases in few different ways. The operation shares some similarities with the scale out operation, in that the operation removed the node that was added to the different database clusters by the scale out operation. Among the databases MySQL stands out in that it experienced seemingly no impact on any of the metrics other than the natural drop in throughput when the slave was terminated.

Figure 13a shows the results from scaling in the CockroachDB cluster. As can be seen, the
operation clearly impacted both the throughput and the latency in a negative way. At the start of the scale in, the data was rebalanced from the fourth node before it was removed from the cluster. The data-rebalancing process took about nine minutes, and it was the reason for the increase in both the network receive rates and the network transmit rates of the nodes. The CPU usage of the first three nodes increased during the operation while the CPU was decreased for the fourth node.

Figure 13c and 13d depicts the results for the TiDB server cluster as well as the TiKV server cluster. As can be seen, the impact of scaling in the TiDB server cluster is very similar to the results obtained from the scale out operation, but reversed. As for MySQL, the drop in throughput when the node was terminated was expected. At the start of the operation when scaling in the TiKV cluster, as for CockroachDB, the data in the cluster was rebalanced which was the reason for increase of the network transmit and receive rates, and the CPU usage of the nodes. The data-rebalancing process took roughly three minutes. As can be seen, during the data-rebalancing process, the client performance was negatively effected.
4.3 Scale up

Figure 14 displays the results from performing the scale up operation on the different databases. The use of the rolling update pattern can be seen across the different databases, where the database nodes (pods) are terminated in reverse ordinal order. The expected result after the operation is higher throughput. This improvement is apparent in MySQL, CockroachDB, and when scaling up the TiKV cluster. Some notable results include the impact on the throughput and latency for CockroachDB and the lack of impact when scaling up the PD server cluster. Furthermore, it is interesting to observe that after scaling up MySQL, the cluster could serve more client transactions while maintaining the same CPU usage as prior to the operation.
Notably, the drop to zero in throughput and latency when scaling up the TiDB server cluster, as seen in Figure 14d, was due to the metric collection frequency in the clients. The drop in throughput during the operation was expected for all databases, because when a node was terminated that had a client instance connected to, the client could not query the databases which consequently resulted in a lowered throughput.

(a) CockroachDB

(b) MySQL
Figure 14: Impact of scale up operation on client performance and server resource usage for CockroachDB (a), MySQL (b), and TiDB (c, d, and e)
4.4 Scale down

Figure 15 shows the results from performing the scale down operation on the different databases. As can be seen, the results are essentially the results from the scale up operation but inverted. However, there are a few differences that can be seen such as in Figure 15c which shows the results from scaling down the PD cluster. When scaling down the PD cluster there was some impact on the performance where the throughput dropped for few moments. There are no real other notable results from performing the operation. The drop to zero throughput for MySQL was due to the same reason as for the TiDB server cluster mentioned in the scale up operation.

The dip in throughput while scaling down the PD server cluster was caused by a slight delay between the termination of the node that was the leader of the cluster and the appointment of a new cluster leader.

![CockroachDB graphs](image1)

![MySQL graphs](image2)
Figure 15: Impact of scale down operation on client performance and server resource usage for CockroachDB (a), MySQL (b), and TiDB (c, d, and e)
4.5 Version upgrade

Figure 16 shows the results for version upgrade. As can be seen in the figure, the behaviour and impact of the operation across the databases are again very much like those for the scale up and scale down operations. This is as expected, as version upgrade is performed in the same rolling update manner as the operations mentioned above. What differentiates version upgrade from scale up/down is the fact that the database nodes are not rescheduled onto new Kubernetes nodes. In contrast, new images for the database containers are pulled before the pods are restarted on the same node. There are no other notable results than those already mentioned for the scale up and scale down operations.
Figure 16: Impact of version upgrade operation on client performance and server resource usage for CockroachDB (a), MySQL (b), and TiDB (c, d, and e)
4.6 Backup

Figure 17 shows the results from performing the backup operation on the different databases. As can be seen, the operation effected the databases in different ways. The most notable result is the lack of impact on the client performance in MySQL, see Figure 17b. The increase of the network transmit rate for mysql-1 was because the node streamed the backup to Google Cloud Storage (GCS) at a rate of about 10 MBps. The behaviour for TiDB and CockroachDB are quite similar, both had a negative impact on the client, while also consuming more CPU and network bandwidth during the backup.

In Figure 17c, the pod that the TiDB operator creates when performing a backup is shown in blue in the network transmit rate diagram. As can be seen, the data from the TiKV nodes was streamed to tidb-1, which in turn streamed the data to the backup pod that stored the data on disk before streaming it to GCS.

In Figure 17a, it can be seen that the node, cockroachdb-0, responsible for performing the backup, had increased CPU usage while receiving the data that was streamed from the other nodes in the cluster.

It is worth noting that the backup on MySQL suffered from the same problem as in the scale out operation, where the operation would fail unless the number of write transactions on the master was capped.
Figure 17: Impact of backup operation on client performance and server resource usage for CockroachDB (a), MySQL (b), and TiDB (c)
4.7 Summary

This section summarizes the experimental evaluation. The operations were evaluated by first and foremost considering the impact they had on the client followed by the impact on the server side metrics. The operations have been given a rating of either:

- **Bad** - Indicating that the operation resulted in unexpected behaviour and/or had a significant impact on both client performance and resource usage for the database.
- **OK** - Indicating that the impact of the operation was as expected, with some impact on both client side and server side.
- **Good** - Indicating that the operation worked as expected, and had a very low impact on both client and server side metrics.

Table 4 shows the summary of the results where each operation for every database has been evaluated and given a rating. As can be seen, the operations scale up/down and version upgrade have been given the same rating due to the similarity of the results obtained for all databases. Furthermore, the scale out and scale in operations were also given the same rating due to the similarities in the obtained results. Overall, MySQL was experienced to have the least impact on both client metrics and server metrics across all operations. Operation of CockroachDB was experienced to have the largest impact on client metrics and server metrics, in particular when scaling up/down and upgrading version. It is worth to note that as all experimental results were qualitative, an interpretation was performed to evaluate the operations for the different databases.

<table>
<thead>
<tr>
<th>Database</th>
<th>Backup</th>
<th>Scale up/down &amp; Version upgrade</th>
<th>Scale out/in</th>
</tr>
</thead>
<tbody>
<tr>
<td>CockroachDB</td>
<td>OK</td>
<td>Not good</td>
<td>OK</td>
</tr>
<tr>
<td>MySQL</td>
<td>Good</td>
<td>Good</td>
<td>Good</td>
</tr>
<tr>
<td>TiDB</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
</tbody>
</table>
Chapter 5

Conclusion

This chapter discusses the fulfillment of the thesis questions posed in Section 1.1. The answers are based on the results from the experiments. Finally, some future work is presented along with some concluding remarks.

• How large is the gap between Kubernetes capabilities and the requirements of stateful database services?

Based on the experiments and the information on Kubernetes in Section 2.2, Kubernetes does have the ability to deploy and run databases using, for example, the StatefulSet. While the StatefulSet provides a set of guarantees that are not necessary, but useful, for deploying databases, it does not capture any domain-specific knowledge on how to correctly operate databases, for example, when scaling out. Operators closes this gap between Kubernetes and databases by implementing the domain-specific operational logic. While operators are not native to Kubernetes, they are an extension and can therefore be considered as a part of Kubernetes. With operators, much of the operational complexity that would normally have to be manually handled by a human with expert knowledge is taken care of by Kubernetes. As the experiments show for CockroachDB, it is possible to run a database in Kubernetes without the use of an operator but it does require domain specific knowledge to be done correctly.

To summarize, Kubernetes has the native capabilities to deploy and run databases but not to fully operate correctly. For the databases where a well implemented operator that handles most, if not all the operational complexity, the gap between Kubernetes and database services is virtually non-existent. For those databases that lack a well functioning operator, the gap is as large as the operational complexity of the database.

• How does different database solutions behave while being hosted in Kubernetes and operated on, and what is the impact of operating on both client performance and resource usage of the database?

The results from the experiments showed how the considered database solutions behaved while being operated, and the consequent impact on both client performance and resource usage of the databases. As the results showed, both the behaviour and the impact was different between the databases. The scale up, scale down, and version upgrade operations all gave very similar results, which was due to the fact that these operations all used the rolling update pattern. What is clear when looking at the results is that MySQL behaves significantly different than the NewSQL databases, where both CockroachDB and TiDB shares similarities with each other. This can be seen by looking at the results from the scale
out and scale in operations. When scaling out and in CockroachDB and the TiKV cluster, data in the clusters was rebalanced to accommodate the change. This behaviour did not exist in MySQL, that merely maintained complete copies of the data set across all nodes. The lack of data distribution and consequently the lack of coordination between the nodes, was what made MySQL excel, especially in the scale out/in and backup operations when compared to both TiDB and CockroachDB. From the results for the scale out/in and backup operations, it is apparent that when data was rebalanced and migrated in both CockroachDB and TiDB, the CPU usage for the nodes was increased, which ultimately had a negative impact on the client performance.

5.1 Future work

This thesis showed that it is possible to host a set of databases and successfully operate them in Kubernetes using the available tools. This thesis also showed and evaluated how well a set of operations worked by looking at how the different databases behaved during operation in Kubernetes, and what the impact was on the client performance and the resource usage of the databases. It is the author’s opinion that the next step logical would be to continue this work by finding a suitable way to interpret the results from this thesis into an answer for how suitable the different database solutions are to being hosted inside Kubernetes. Another interesting area that could be further explored is to evaluate different operators for the same database solution. For some databases there exists multiple different operators, for example, for MySQL there exists five operators[^1].

Furthermore, the thesis only considered a small set of SQL and NewSQL databases. There exists a multitude of other databases such as PostgreSQL[^2], Oracle[^3] and Microsoft SQL Server[^4]. In addition to SQL and NewSQL databases, there is also a large group of so-called NoSQL databases such as Cassandra[^5] and MongoDB[^6]. The NoSQL databases makes the claim to be highly scalable and might therefore be suitable for being hosted in Kubernetes.

[^1]: https://github.com/operator-framework/awesome-operators
[^2]: https://www.postgresql.org/
[^3]: https://www.oracle.com/se/database/
[^5]: http://cassandra.apache.org/
[^6]: https://www.mongodb.com/
References


Appendix A

Miscellaneous

A.1 SysBench configuration

A.1.1 MySQL

Listing A.1: SysBench configuration for instance with write-only transaction.

```bash
--threads =4
--db-driver =mysql
--mysql-host =xxx
--mysql-user =xxx
--mysql-password =xxx
--mysql-port =3306
--report-interval =10
--tables =25
--table-size =1000000
--db-ps-mode =disable
--mysql-storage_engine =innodb
--thread-init-timeout =1
--time =0
--rate =40

Script: oltp_write_only.lua
```

Listing A.2: SysBench command line arguments for read-only transactions.

```bash
--threads =4
--db-driver =mysql
--mysql-host =xxx
--mysql-user =xxx
--mysql-password =xxx
--mysql-port =3306
--report-interval =10
--tables =25
--table-size =1000000
--db-ps-mode =disable
--mysql-storage_engine =innodb
--thread-init-timeout =1
--time =0
--rate =0

Script: oltp_read_only.lua
```
A.1.2 TiDB

Listing A.3: SysBench command line arguments used for TiDB.

```bash
--threads=4 \
--db-driver=mysql \
--mysql-host=xxx \
--mysql-user=xxx \
--mysql-port=4000 \
--report-interval=10 \
--tables=25 \
--table-size=1000000 \
--mysql-password=xxx \
--db-ps-mode=disable \
--time=0 \
--mysql-ignore-errors=1105 \
--thread-init-timeout=1 \
```

Script: oltp_read_write.lua

A.1.3 CockroachDB

Listing A.4: SysBench command line arguments used for CockroachDB.

```bash
--threads=4 \
--db-driver=pgsql \
--pgsql-host=cockroachdb-0.cockroachdb \
--pgsql-user=sbtest \
--pgsql-port=26257 \
--pgsql-db=sbtest \
--report-interval=10 \
--tables=25 \
--table-size=1000000 \
--db-ps-mode=disable \
--thread-init-timeout=1 \
--time=0 \
--rate=0 \
--skip_trx=off \
--auto_inc=off \
```

Script: oltp_read_write.lua

A.1.4 Table definition

Listing A.5: Table definition for the tables used.

```sql
id INTEGER NOT NULL AUTO_INCREMENT,
k INTEGER DEFAULT '0' NOT NULL,
c CHAR(120) DEFAULT '' NOT NULL,
pad CHAR(60) DEFAULT '' NOT NULL,
PRIMARY KEY (id)
```

A.2 cAdvisor

Kustomize was used to patch the original cAdvisor daemonset with the desired arguments. The following Kustomize was used:

```bash
kustomize_2.0.2_linux_amd64
```

and the cAdvisor image was:

```bash
k8s.gcr.io/cadvisor:v0.30.2
```
Listing A.6: cAdvisor arguments, cadvisor-args.yaml

```bash
--housekeeping_interval=5s
--max_housekeeping_interval=15s
--event_storage_event_limit=default=0
--event_storage_age_limit=default=0
--disable_metrics=percpu,tcp,udp,disk
--docker_only
```

For more information on how to build see: https://github.com/google/cadvisor/tree/master/deploy/kubernetes

A.3 MySQL error

InnoDB: Number of pools: 1
xtrabackup: error: log block numbers mismatch:
xtrabackup: error: expected log block no. 13393709, but got no. 13590309 from the log file.
xtrabackup: error: it looks like InnoDB log has wrapped around before xtrabackup could process all records due to either log copying being too slow, or log files being too small.
xtrabackup: Error: xtrabackup_copy_logfile() failed.
2019-03-28T11:20:30.817Z ERROR sidecar failed waiting for xtrabackup to finish "error": "exit status 1"