Using cloud virtualization technologies for basic database operations

Author:
Sanjaykumar Reddy Beerelli

December 3, 2018

Advisors:
M.Sc. Gabriel Campero Durand
Data and Knowledge Engineering Group

Prof. Dr. rer. nat. habil. Gunter Saake
Data and Knowledge Engineering Group
Beerelli, Sanjaykumar Reddy:
Using cloud virtualization technologies for basic database operations
Abstract

With increasing amounts of data, database systems are called upon everyday more to optimize the runtime and resource consumption of queries. To accelerate database workloads, there are some basic alternatives, like scaling out the computing such that other processing devices are used, or scaling-up by employing specialized hardware features of a device in use, for example, SIMD instructions or multi-threading or exploiting additional multi-core processors and heterogeneous co-processors (e.g., graphical processing units). By leveraging parallel processors and special hardware features, the performance of database systems can be reasonably improved.

With the development of cloud technologies, both choices of scaling-up and scaling-out database deployments can be tackled in innovative ways. On one side, hardware sensitive features can be used through container-based processing, which aids the deployment of a database process over different hardware available, but introduces a level of indirection (with the virtualization) over such hardware. Similarly, the distribution of processing can now also be managed with serverless computing, an approach in which the management of processes and threads is left to a virtualized cluster manager, and not to the operating system.

In this Thesis, we provide some early evaluations of how these two approaches could be leveraged for data management. In specific we research on how serverless functions might be used to scale database clients for transactional workloads, and the potential improvements available by using auto scale-up features. We also study and report on the impact of virtualization on the execution of specialized co-processor code.

In order to study serverless technologies we select Google Cloud Functions as a serverless framework, Redis, a popular key-value store, as a database system, and the Yahoo Cloud Serving Benchmark (YCSB), as a workload. We implement a serverless YCSB client for Redis, studying the role of clients and configurations in influencing the performance of the serverless functions with respect to that of a general Redis YCSB client. Among our findings, from evaluating on a desktop computer and on Google Cloud, we find that serverless functions with local cloud emulators can match and outperform the throughput of traditional deployments for data ingestion into Redis; while read operations are still better served without serverless processing. We can also report that, counter-intuitively, when migrating to a cloud provider with basic settings, serverless processing seems to lose its competitive edge for data loading.
Regarding the virtualization of hardware-sensitive features, we study the impact of container deployment for small CUDA GPU samples by using NVIDIA-Docker. We report small differences in performance, with some container samples performing slightly better when compared to the host execution; for samples that require kernel services, container performance decreased, but not by a large margin. Thus we can report the interesting outcome that specialized hardware features are able to be executed from within containers, without affecting the expected performance. Our findings indicate that there can be expected little performance overheads in migrating hardware-specialized databases to cloud-based platforms.

We expect that this work can help readers to understand better how container virtualization works for hardware-sensitive features, and how serverless functions could be adapted such that they benefit database operations.
Acknowledgements

By submitting this thesis, my long term association with Otto von Guericke University will come to an end.

First and foremost, I am grateful to my advisor M.Sc. Gabriel Campero Durand for his guidance, patience and constant encouragement without which this may not have been possible.

I would like to thank Prof. Dr. rer. nat. habil. Gunter Saake for giving me the opportunity to write my Master’s thesis at his chair.

It has been a privilege for me to work in collaboration with the Data and Knowledge Engineering Group.

I would like to thank my family and friends, who supported me in completing my studies and in writing my thesis.
vi
Declaration of Academic Integrity

I hereby declare that this thesis is solely my own work and I have cited all external sources used.

Magdeburg, December 3rd 2018

__________________________________________
Sanjaykumar Reddy Beerelli
## Hardware sensitive features

4.1 Implementation - The third step ........................................ 31
4.2 Evaluation Questions .................................................. 31
4.3 Implementation ......................................................... 32
  4.3.1 Native system execution ........................................... 32
  4.3.2 Virtualization of hardware-sensitive features ................. 32
4.4 Evaluation ............................................................... 33
  4.4.1 asyncAPI ............................................................ 33
  4.4.2 SimpleMultiCopy .................................................... 34
  4.4.3 Bandwidth Test ..................................................... 34
  4.4.4 Blachscholes ....................................................... 35
4.5 Summary ................................................................. 36

## Serverless Computing for databases

5.1 Implementation - The third step ........................................ 39
5.2 Evaluation Questions .................................................. 39
5.3 Implementation ......................................................... 40
  5.3.1 Native System Execution ........................................... 41
  5.3.2 Cloud Emulator execution ......................................... 42
  5.3.3 Cloud platform execution .......................................... 43
  5.3.4 Native execution in compute instance ............................ 45
5.4 Evaluation ............................................................... 45
  5.4.1 YCSB data Load ...................................................... 45
  5.4.2 YCSB data run ..................................................... 49
5.5 Summary ................................................................. 51

## Conclusion and Future Work

6.1 Summary ................................................................. 55
6.2 Threats to validity .................................................... 57
6.3 Future work ............................................................. 58

## Appendix

7.0.1 Implementation code ................................................ 61
7.0.2 Python flask file for native execution ............................ 61
7.0.3 YCSB file for Read, Insert, Update and Scan ................... 63
7.0.4 Serverless function .................................................. 66

## Bibliography

69
List of Figures

1.1 Waterfall model with different phases ........................................ 4

2.1 Report from Google Trends for term “serverless” .......................... 11

2.2 Serverless platform architecture [BCC+17] .................................... 12

4.1 Comparison between normal execution and virtualized execution of hardware sensitive features for asyncAPI ............................ 33

4.2 Comparison between normal execution and virtualized execution of hardware sensitive features for SimpleMultiCopy ....................... 34

4.3 Comparison between normal execution and virtualized execution of hardware sensitive features for Bandwidth test ........................... 35

4.4 Comparison between normal execution and containerized execution of hardware sensitive features for BlackScholes test ....................... 36

5.1 Implementation ............................................................................ 40

5.2 Using redis-cli command to connect to the redis-server ................ 41

5.3 Snapshot of 10000 keys loaded into Redis .................................... 44

5.4 Throughput of YCSB Load operation for all executions ................. 46

5.5 Throughput of YCSB Load operation using serverless function in Google cloud ................................................................. 47

5.6 Average latency of YCSB Load in different executions .................. 48

5.7 Serverless function Log file ....................................................... 49

5.8 Throughput comparison of YCSB run for different executions ........ 50

5.9 Latency for Workload-A ............................................................ 51

5.10 Latency for workload-B ............................................................ 51
1. Introduction

In this chapter, we will present the motivation behind the thesis, describe its goals and outline its organization.

Nowadays databases are commonly used in every organization. Since data volumes are increasing drastically, database systems are required to be fast and efficient, scaling beyond a single processing node. However the management of system scale-out is not always trivial, as different nodes might require manual initialization and configuration of the database node. Furthermore different nodes might have different operating systems and different versions of supporting tools.

One common solution to facilitate the process is the use of virtual machines, which can offer a standard configuration over different compute nodes. But this solution does not help performance so much, because these systems use hardware virtualization, which could impede or degrade the use of specialized hardware features. Therefore applications that require good performance cannot rely on them. Furthermore, managing database servers with hardware-level virtualization (i.e., by running the database within a virtual machine) can be cumbersome as the database resources have to be shared among various virtual machines.

To improve the performance with virtualization, Operating system(OS)- level virtualization using containers can be done. This is also known as Containerization. Containers are light-weight, with less start-up time compared to a virtual machine. With containers, OS-level virtualization is used; in this approach not the hardware instructions, but the operating system calls are virtualized. Containers offer virtualization with close to no overhead respect to direct execution, when compared to VMs [SPF+07, FFR15]. Containers can also be managed with a cluster manager. Examples of cluster manager are Kubernetes, Apache Mesos and Docker Swarm. With the adoption of container technologies and cluster managers, another solution currently being used is Serverless computing.
Serverless computing is a recent technology that started to gain importance in cloud technology. It facilitates the execution of lightweight functions with self-scaling features and asynchronous execution, with the scheduling and deployment handled by the cluster manager. This approach is also referred as Function as a Service (FaaS).

Both OS-level virtualization and serverless computing are, relatively, in early stages of research. To date, and to our knowledge, there is no study on how these could be used for database systems. Such studies are relevant to ease the adoption of the technologies, helping the maintenance of databases and exploiting cluster management-based scheduling of database tasks.

One limitation in the adoption of these technologies for database purposes is the lack of research on their applicability. Specifically, it is not clear to what extent serverless functions can improve database calls by scaling, for example. Neither are there studies covering the impact of configurations on the performance of serverless functions. In addition, regarding serverless computing, it is not clear if there are opportunities for it to benefit complex resource-intensive database operations, like analytical tasks; or be used in communicating transactional updates to analytical processes in hybrid transactional analytical processing.

From our research, we would like to consider whether serverless functions can be used effectively for scaling database calls. We would also like to study the difference of using serverless functions in a local machine when contrasted to a cloud system.

Furthermore, since databases use specialized features from hardware, it is not clear if container technologies could have an impact or not, on the performance; since they could introduce overheads, and they have different scheduling approaches than those of basic operating systems.

Both of these research gaps limit the benefits that cluster managers could bring to database maintenance, leading to wasted opportunities.

Though there is a body of research comparing VMs against containers for several scenarios, including how they fare for interfering neighbors (i.e., when neighbors are co-located in the same processing device), and, additionally, there is work on designing OS-structures to better isolate containers running on a single OS; [RF18] to our knowledge there is little current work on the intersection of databases and virtualization.

There is some research work done on comparing both hardware virtualization and container virtualization when these techniques are run on a CPU. Specifically authors show that pinning a container to logical cores can lead to better performance for databases, when compared to automatic cluster management or OS core selection. Authors also evaluate the impact of multiple tenants on a single system, showing that for containers the impact is higher than for VMs [RF18]. Similar work was done by Mardan and Kono, who show that shared OS structures, such as the journaling subsystem, can deteriorate the performance of DBMSs running on containers over that of DBMSs running on VMs [MK16].
1.1. Research aim

From the research done by Qi Zhang et al. for the study on virtual machines and containers in a big data environment it is shown that containers are more convenient in deployment and boot-up. For big data workloads, much better scalability is obtained compared to virtual machines. On a same workload, authors show that containers achieve higher memory and CPU utilization\cite{ZLP*18}.

Thus in our work, we intend to address both research gaps. First, we propose to evaluate the impact of virtualization on different general-purpose GPU samples, like the Nvidia-Cuda samples, to compare the throughput and operational timings by containerizing hardware-sensitive features(GPU) with Docker containers against traditional execution.

Second, we evaluate the applicability of serverless functions. Recent advancements and the popularization of container technologies contributed to the emergence of the novel serverless approach.\cite{BCC*17} With a standard Yahoo Cloud Serving Benchmark(YCSB) benchmark using a Redis database, we propose to study the performance of serverless functions for improving database calls. For this we develop a YCSB benchmark Redis client using Google Cloud Functions. Our tests are run to compare the throughput and latency of the YCSB benchmark when running on the Google Cloud Emulator (GCE) versus normal execution, and also compared to execution on the Google Cloud platform.

By the end we evaluate the applicability of containers to support hardware-sensitive features, and of serverless functions to improve database calls. Further studies could continue our research, for example by distributing co-processor accelerated systems using container technologies and studying the impact of noisy neighbors and file system sharing on the goodness of the system vs. that of VM deployments; or by employing serverless functions for further database processes, with more studies into the role of the cluster management technologies, characteristics from the serverless offerings of vendors and better adopting event processing.

1.1 Research aim

We propose the following research questions to serve as focal points for our work:

1. Container technology: Can hardware-sensitive features be used successfully after virtualization with containers? What is the throughput compared to normal execution? Is there an overhead from the virtualization?

2. Serverless computing: Can serverless functions support basic database operations? If so, what is the performance observed when compared to basic execution? Can serverless functions be used to automatically scale-up the processing? What is the throughput comparison using a real-time cloud platform service? Can the cloud emulator performance be replicated in the cloud platform?

1.2 Research methodology

To develop, design and test a software product of high quality, within the scope of research, a Software Development Life Cycle(SDLC) is required. Different models have
been defined and designed for software development. Each process model has its own unique steps to organize software development such that the end products are successful. These models are self-reliant on tools and technologies. To find out the answers for the above-mentioned research questions, we have selected to rely on the Waterfall model. Every research question, in-turn, has all the phases that are present in the waterfall model [Roy87].

The earliest approach for software development was done using Waterfall model. It is also known as the linear-sequential life cycle model.

![Waterfall model with different phases](image)

Figure 1.1: Waterfall model with different phases

Figure 1.1 shows the sequence of steps in a software development. The process of software development is divided into separate phases. The output of one phase acts as an input to the next phase. The phases are described below:

- **Requirements**: In this phase, the requirements of the system to be developed is selected. The aim is to find out the goal to be achieved. A clear idea of what is required and what can be achieved is needed. If this step is neglected, the whole process leads to undesired results which waste engineering efforts. For the case of our work, in this stage we studied the background for our research, and we defined the research questions to address.

- **Design**: In this phase, the requirement specifications from phase one are studied and a design is prepared. In the waterfall model, there is the assumption that once the design is decided upon, it will be used without changes until the end of the iteration. For our work, in this step we defined how the implementation and evaluation should be done.

- **Implementation**: In this phase, analysis, coding, and testing is done. Depending on the output from the design phase, the resources are allocated and the experimental setup is done. The system is developed in small units. Testing of the
developed units is done in this phase. As the testing phase is at the end of the software development lifecycle, the programmer has to take good care in design and implementing. Any error in early stages could yield to massive waste of resources and time. For our work, the implementation phase consisted of implementing and configuring the software required for our evaluations.

- **Verification:** In this phase, we evaluate how close the practical results are with the theoretical approach. All the reasons that are responsible to make the model inadequate to meet the requirements are determined. For our work this phase consisted on running our experiments, analyzing the findings and, finally, documenting our work.

- **Maintenance:** In this phase, the data obtained from previous phases are put together and released it to the clients. Maintenance is often required in the client environment. New versions of the product are released to enhance the performance. Due to the nature of our Thesis project, there are no maintenance tasks performed.

### 1.3 Thesis structure

The thesis is structured as follows:

- **Technical Background** provides an overview of current research work such as hardware-sensitive features and hardware virtualization and its techniques. We also discuss the state of the art of serverless computing and serverless clients ([Chapter 2](#)). This chapter serves as an artifact from the requirements phase.

- **Prototypical Implementation** documents the prototypical implementation of the models used for the research work. We discuss the evaluation questions and the experimental setup ([Chapter 3](#)). This chapter serves as an artifact from the requirements phase.

- **Hardware-Sensitive features** We evaluate how hardware-sensitive features perform under containerization ([Chapter 4](#)). This is the first evaluation question that is solved with the Waterfall model. The chapter covers the implementation and verification phases.

- **Serverless Computing for databases** includes our concept for implementing a serverless computing functionality to support calls to a database. We compare experimentally the serverless functions throughput with normal execution throughput for a YCSB benchmark ([Chapter 5](#)). The chapter covers the implementation and verification phases.

- **Conclusion and Future Work** conclude our work by summarizing our study and findings. We disclose in this section with threats to the validity, and future scope of our work. ([Chapter 6](#))
1. Introduction
2. Technical Background

In this chapter, we present an overview of the theoretical background and state of the art relevant to the current research work. Since our work is on cloud virtualization and serverless computing which is still in development, in this chapter we do not attempt to provide a comprehensive survey on them. Instead, we carry out a focused research, providing sufficient information for understanding the context of our research, and presenting with care the main ideas necessary for understanding our research questions and focus. We outline this chapter as follows:

- In (Section 2.2), we discuss the concept of virtualization and different virtualization types.
- In (Section 2.2.2), we discuss in brief about containers and Docker, a popular container software.
- In (Section 2.3), we discuss in detail about serverless computing and its architecture, and applications. We discuss in brief the available cloud platforms and aspects of serverless computing.
- In (Section 2.4), we discuss in detail the performance of virtualized systems in general, in database management systems, and in hardware-sensitive features virtualization.

2.1 Requirement Analysis - The First step

In our work, to analyze the requirements we followed first step of the waterfall model. From literature research and by examining relevant technical background these requirements are observed. The study of background and literature research is given in the below sections.
2. Technical Background

2.1.1 Literature research

In this section, we present an outline of the process followed for the literature research.

• In the basic search phase, we focused on articles that are relevant to virtualization and serverless computing in general. We used the Google Scholar database to search the literature papers. In this phase, no extensive study of the papers was done to select the relevant topic.

For hardware sensitive features, the search term used are:

– “virtualization performance”, “GPU virtualization” and “virtual machines vs containers”. The literature was selected in a time period that lies between 2007-2018, corresponding to the development of the technologies.

For serverless computing, we searched using:

– “serverless computing for databases”. We selected the literature papers from 1-10 pages from the search results. Sadly, we couldn’t find any literature that is relevant to serverless computing for databases. But we considered the literature papers that talks about the state-of-art and the application of serverless functions. The literature is selected in a time period between 2016-2018, corresponding to the development of serverless technologies.

In the detailed search phase, we exclude the literature papers from the first phase that were not found to be relevant to our research topic. If a paper is a bachelors or a master thesis, unpublished or labeled as work in progress, it was excluded. From the obtained resources, new search terms were acquired which we followed to more relevant articles. After the detailed study of all the collected sources, with complete study of their bibliography, the 31st most relevant literature sources were selected. We base our study on them.

2.2 Hardware virtualization

Virtualization creates an abstraction of computing resources. Virtualization can be defined as the act of creating a virtual version of computing infrastructure like network resources, hardware platforms. Virtualization benefits computer infrastructure by adding flexibility and agility. Databases these days are mostly run in virtualized environments. Virtualizing database components involve server virtualization that converts a data-center into an operating cloud. Server virtualization helps to improve cluster elasticity and Utilization of shared servers is enhanced.
2.2.1 Virtual Machines (VM’s)

A virtual machine is created using a Hypervisor or a Virtual Machine Monitor (VMM). A virtual machine introduces an abstraction between virtual resources and physical resources. A Virtual machine works as a real computer with a guest OS, however it can be deployed on any other physical machine. To meet service requirements, multiple VM’s can be started and stopped on demand using a single physical machine. The task of deciding on which server to run a VM is also important for managing large-scale VM-based applications. This is called server consolidation. A physical database server can be virtualized into several Virtual Machines (VMS).

There are three kinds of virtualization techniques

**Full virtualization**

In this method, host hardware is completely transformed into a virtual CPU, virtual memory for use by the virtual machine using its unmodified Operating system.

**Partial virtualization**

As the name suggests, some host resources are virtualized and some are not. The guest programs must be modified to run in such an environment.

**Container-based virtualization**

The concept of this technique is quite similar to the one with the hypervisors, but it is implemented in a different way. Libraries and executables are shared among the containers. The hardware of the system is not virtualized as the containers share the same kernel that manages resources of the system. This approach can significantly reduce the overhead that is seen in hypervisors by removing the redundant kernel level resources \(^{[SPF+07]}\).

In order to develop an application that requires five micro-services in a single machine, five virtual machines are needed which wastes a lot of resources. Containers provide a better solution with efficient use of resources and better performance.

2.2.2 Containers

Containerization is an Operating System (OS) level virtualization. There are different kinds of containerization software, among them Docker is a popular container software. The applications that are built in Docker are packaged with all the supporting dependencies into a standard form called a Container \(^{[RBA17]}\), the instructions to build a container are specified in a single file with a standard language for it, and they can be made public and are kept in repositories such as Docker Hub. Docker containers allow to build, ship, test and deploy applications in a lightweight packaging tool known as the Docker Engine. In containers, applications are virtualized and run. Containers can
provide a consistent computing environment through the whole software development life cycle (SDLC), and through the use of build files, they facilitate the management of configurations.

Dockerfile, Docker image and Docker hub are three main components for a Docker container. Docker hub is a cloud-based registry service that links code repositories. Docker hub contains official repositories where base images are updated regularly and can be used to develop new images. A Docker image that is built can be uploaded to Docker hub. A Developer writes code for an application with requirements needed in a Docker file. A Docker image is built based on the docker file written by the developer. A docker file should have a base image to build on. A Docker container is built from one or more Docker images. A Docker container consists of run-time instances of a Docker image. A Docker container is an isolated platform. A Container has everything needed to run an application.

2.3 Serverless computing

Cloud computing is a modern form of information systems management. Cloud computing provides users with IT resources just by paying a fee, without the need to own servers. As resources are used on-demand, running costs are reduced. Cloud computing provides many advantages for enterprises and organizations. There are three basic and well-known services in cloud computing: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software-as-a-Service (SaaS) [Kra18]. In the Infrastructure-as-a-Service (IaaS) model both the application code and the operating infrastructure in the cloud are controlled by the developer. Here, the provisioning of hardware or virtual machines is done by the developer. Every application that is deployed and executed in the IaaS model is taken care of by the developer. In PaaS and SaaS models, the developer does not manage the infrastructure and has no control over it. Instead, pre-packaged components or full applications can be accessed by the developer. The code is provided by the developer, though the execution of the code is bound to the cloud platform, either by using run-times (e.g. Java VMs, containers, or Cloud Foundry Buildpacks, which pre-package run-times of different languages), or by using underlying software systems (e.g. cloud-hosted databases, or Watson Services in IBM Bluemix) [BCC+17].

Serverless computing is also known as Function-as-a-service (FaaS). It is developed as a new paradigm for cloud applications deployment. This is mainly made possible by the development of container technologies, and the popularization of micro-service architectures in enterprise applications. Figure 2.1 shows the Google Trends report on increasing popularity of the term “serverless” in the last five years. This shows the increasing attention to serverless computing in the development community and industry trade-shows.
In serverless computing, the code is written in the form of stateless functions. The developer is not concerned about deployment and maintenance of code. The code written is expected to be fault-tolerant, and capable of exposing logic for auto-scaling (e.g. if the code serves an HTTP request, it can be scaled as the number of requests grow, with the developer providing rules for how much the code can scale). No servers will run when the user function code is idle, and the user doesn’t need to pay for VMs or expensive infrastructure during these situations. Such a scenario is unlikely, in Platform-as-a-Service, where the user would, by default, be charged even during idle periods [BCC+17].

### 2.3.1 Generic Serverless Architecture

There is a common misunderstanding about the term “serverless”. Servers are naturally needed, but developers don’t need to worry about managing them. Serverless platforms take care about decisions such as defining the number of servers and server capacity, according to the workload.

Architecturally, serverless platforms must contain an Event processing system, which serves to the fundamental ability of serverless platforms to run codes based on trigger events, as shown, generically in Figure 2.2. This is a generic architecture, and real platforms might differ in the exact constituent components.

The user functions (code) are registered with the cloud serverless provider. Based on the events from an event source, the registered functions could be triggered. First, events, such as a user access to an HTTP endpoint, are enqueued, such that events can be managed as a group. Here triggers are expected to be sent over HTTP or received from an event source (e.g, a message bus, like Kafka). For each event, the serverless system must identify the function that is responsible to handle an event.

Next, events are dispatched based on the resources available. In Figure 2.2, the dispatcher starts worker processes related to each event. Worker processes are like sandboxes or containers where the function runs, they are also called function instances. The execution logs should be made available to the user. Usually, the platform does not need to track the completion of functions. The function instance is stopped when it is no longer needed.
Implementing such functionality by considering cost, scalability and fault tolerance is a challenging task. A serverless platform must be quick and efficient to start a function and to process its input. The platform needs to enqueue events, depending on the state of queues and rate of event arrival, execution of functions needs to be scheduled, stopping and deallocating resources for idle function instances has to be managed. Scaling and managing failures in a cloud environment has to be effectively handled by the serverless platform [BCC+17].

2.3.2 Applications

In this section we collect relevant examples of serverless applications.

Serverless computing is used in processing background tasks of Web and Internet of Things applications, or event-driven stream processing [MGZ+17].

Serverless computing is used in different scenarios that include Internet of Things with fog computing [PDF18] and edge computing [BMG17], parallel data processing [JPV+17], and low latency video processing [FWS+17].

Serverless architecture is also used for large-scale analytical data processing using Flint, a spark execution engine prototype that works along with Amazon AWS Lambda. With the help of Flint, a Spark Cluster is not needed, instead, a PySpark can be used transparently and jobs run only when needed. The results show that big data analytics is viable using a serverless architecture [KL18].

Authors have proposed Snafu, an open-source FaaS tool which allows managing, executing and testing serverless functions of different cloud platforms. Snafu imports services from Amazon AWS Lambda, IBM Bluemix OpenWhisk, Google cloud functions and also provides a control plane to three of them. Snafu supports many programming languages and programming models. Using Snafu authors have tested different scientific computing
experiments with functions which include mathematics (calculation of pi value), computer graphics (face detection), cryptology (password cracking) and meteorology (precipitation forecast). Authors show four different experiments with different computing requirements with respect to storage and resource utilization. For scientific and high-performance computing, simple functions which are executed in self-hosted FaaS platforms are considered as a better solution than running over cloud vendors.\cite{SMM17}

A video job typically needs a lot of CPU. A 4K or a virtual reality video with one-hour runtime takes around 30 CPU-hours to price. Serverless computing is used in processing low latency videos. According to Fouladi et al. \cite{FWS17} a system ExCamera is developed that can edit, transform and encode a video with low latency using serverless functions. The system consists of two important contributions. First, a framework is designed such that parallel computations are run on existing cloud computing platforms. In this system, thousands of threads are started in a matter of seconds. The system also manages communication between them. Secondly, a video encoder is implemented that intends parallelism, using a functional programming such that the computation can be split into tiny tasks without affecting compression efficiency. Amazon AWS Lambda is used as a cloud function service and the functions are written in C++. As the microservice framework executes asynchronous tasks, and video processing requires thousands of threads that run heavy-weighted computations. In order to handle this mismatch, a library (mu) is developed to write and deploy parallel computations on Amazon AWS Lambda. AWS Lambda is selected as a serverless platform because: (1) workers spawn quickly, (2) billing is in sub-second increments, (3) a user can run many workers simultaneously, and (4) workers can run arbitrary executables. By using AWS Lambda cloud functions, many parallel resources can be accessed, started or stopped faster compared to Amazon EC2 or Microsoft Azure that rely on Virtual machines. When tests are made for two 4K movies (animated and live action), ExCamera that uses serverless functions achieved 2% (animated) and 9% (live action) of the performance of state-of-art encoder with a high level of parallelism. Besides commercial serverless platform, there are also some academic proposals for serverless computing. Hendrickson et al. \cite{HSH16} after identifying problems in AWS Lambda proposed OpenLambda, to handle long function startup latency.

2.3.2.1 High-performance computing

According to Ekin Akkus et al., when an application runs on a serverless platform that follows a particular execution path connecting multiple functions, then the serverless platforms don’t perform better due to overheads. The degrading performance in existing cloud platforms is caused by a long startup latency due to cold containers (i.e., each function is executed generally in an isolated container, hence when a function is triggered, the function associated container starts and has to be stopped when the execution of the function is done, which takes time and leads to higher latency when compared to code that does not require such startup), and inefficient resource management. To overcome this problem a novel serverless platform, the SAND system is proposed by authors. It is a new serverless computing paradigm through which authors aim to
support high-performance computing. SAND provides low latency and efficient resource utilization compared to existing serverless platforms. To achieve the mentioned features, SAND follows two techniques 1) Application level sand-boxing (using two levels of isolation. Strong isolation among applications in a sandbox, weaker isolation among functions running in a sandbox) and 2) using a hierarchal message bus (using a local bus and a global bus on each host to make sure the messages are transferred fast, which makes all the functions execution to start instantly). By using these techniques SAND achieves low latency and efficient resource management.

The SAND system consists of the application, grain and workflow. The SAND system is tested with an image recognition system pipeline that contains four executable functions: extract image metadata, verify and transform it to a specific format, tag objects via image recognition, and produce a thumbnail. The serverless functions when running in SAND system are well performed for high-performance computing with some limitations. Main limitations are selecting a sand-boxing system either containers, VMs, Unikernels, Light-weight contexts (LWC), gVisor. Each has their own advantages and disadvantages. Furthermore, hierarchal queuing used in the SAND system can induce sub-optimal load balancing. Another limitation is, using a single host to run multiple sandboxes makes the functions compete among themselves for the resources and impact the performance. Keeping these limitations in mind, the future scope would be to distribute applications functions and sandboxes across hosts such that, better load balancing is achieved with better latency.

**2.3.3 Current platforms and comparisons**

An application in serverless computing consists of one or more functions. A function is a standalone, stateless and small component to handle certain tasks. A function is generally a piece of code written in a scripting language. The execution environments and servers for functions, allocating resources to handle scalability are managed by serverless platform providers. Many serverless platforms are being developed and deployed in recent years, which are most commonly used in many application are Amazon AWS Lambda, Microsoft Azure, Google cloud platform, IBM Bluemix OpenWhisk. A function(code) in all these platforms are run in a container or in a sandbox with a limited amount of resources. A brief discussion of cloud platforms and their comparisons are seen further:

1. **Amazon AWS Lambda**

   It is an Amazon web service for serverless computing. Lambda supports different programming languages that include: Node.js, C#, Java, Python. Trigger events for lambda are uploading an image, website clicks, in-app activities, and other custom requests. It is a public runtime environment with automatic scaling. The Orchestration is done using AWS step functions. A maximum number of 1500 functions can be deployed in a project with max deployment size of 50MB for a single function. The max duration of a function before it is forcibly stopped is 300 sec. Amazon web services are used in many use cases that include data processing (real-time file processing) and server
2.3. Serverless computing

Lambda is highly used in Netflix, Earth Network (sensor data detection, monitoring) and so on and so forth.

2. Microsoft Azure functions

Azure functions are released as a general edition in November 2016. It is an open source runtime environment with manual and an automatic scalability. Azure supports functions written in C#, Node.js, Javascript, Window Scripting, Power shell, Bash, PHP1, Python. Event triggers for Azure functions are HTTP requests, scheduled events, Azure service bus. Information regarding the number of functions and deployment size is unknown in Azure. The max duration of a function before it is forcibly stopped is 600 sec. Azure functions use cases as cited by Microsoft are Software-as-a-Service event processing, mobile backends, real-time stream processing (IoT).

3. Google Cloud Platform

It is released basically for Google cloud services. It is a public runtime environment with the auto-scaling feature. Cloud functions are written in Node.js, Python, JavaScript. Events are triggered using HTTP, Google cloud storage, Google cloud pub/sub. A maximum number of 1000 functions can be deployed in a project with max deployment size of 100MB(compressed) for sources and 500MB for uncompressed sources and modules. The max duration of a function before it is forcibly stopped is 540 sec. Specific use cases for Google Cloud Functions include mobile backend, APIs and micro-service development, data processing/ETL, web-hooks (for responding to third party triggers) and IoT.

4. IBM Bluemix OpenWhisk

IBM Bluemix OpenWhisk is IBM’s serverless cloud computing platform. It was released for general use in December 2016. It is an open source runtime environment with an auto-scaling option. Functions are written in Swift and Javascript. Event triggering is done using HTTP, Alarms and Github webhooks. There seems to be no maximum number of functions that can be deployed in a project. The max duration of a function before it is forcibly stopped is 0.1-300 sec. The most common use cases of OpenWhisk are micro-services, web, mobile and API backends, IoT, and data processing. OpenWhisk can be used in conjunction with cognitive technologies (e.g. Alchemy and Watson) and messaging systems (e.g. Kafka and IBM Messaging Hub). No high profile users could be identified that used OpenWhisk. IBM highlights Docker container integration as a comprehending point from AWS Lambda and Google Cloud Functions.

Amazon web services is most commonly used in both the enterprise serverless cloud computing and also in academic level. There is no discrete academic level research done using Google cloud platform, Azure functions. IBM Bluemix OpenWhisk is used in two papers that deal with event-based programming triggered by different ways like data from a weather forecast, application data from an Apple Watch, and speech utterances [BCC+16]. IBM Bluemix OpenWhisk that provides an IBM Watson services includes news, jokes, dates, weather, music tutor, and an alarm service with help of a chatbot [YCCII16].
According to Lian Wang et al. [WLZ+18], performance isolation and resource management of three popular serverless platforms provided interesting results. Amazon AWS Lambda achieved better scalability and low cold-start latency. The performance isolation lacks among function instances in AWS which causes up to a 19x decrease in I/O, networking, or cold-start performance. In AWS fixed amount of CPU cycles has been allocated to an instance that is based only on function memory. Google platform similar mechanism as AWS but has a median instance of 11.1% to 100% as function memory increases. Azure has high CPU utilization rates compared to other platforms. More results on the performance of Azure, Amazon AWS Lambda, and Google cloud platform can be found in [WLZ+18].

The selection among serverless platforms has to be done based on the requirement of the developers, requiring cost analysis and some practical evaluations for selecting a vendor.

### 2.3.4 Other aspects

Serverless architecture have many advantages when compared to traditional server-based approaches. Serverless architecture can be used with Edge computing to empower low latency applications. According to Baresi et al. a serverless architecture proposed at an Edge outperforms cloud-based solutions. The aim of the research is to show that the serverless edge architectures performs better than a typical serverless cloud provider for low-latency applications. The research was carried out on a Mobile Augmented Reality (MAR) application with an Edge computing solution that used a serverless architecture. The task of the application is to help the visitors who want information relevant to their Points-of-interests (POI) like monuments, architectural elements by looking them through their mobile. The Edge node uses the Openwhisk serverless framework and the cloud alternative used is AWS Lambda. Openwhisk has a built-in NoSQL database: CouchDB which responds to user-defined triggers and rules. The payload used in this experiment is an image of size approximately 500KB. The tests are done for 100 and 1000 requests, where the edge based solution outperformed the traditional serverless application by 80% in throughput and latency for 100 requests, and for 1000 requests the throughput is almost the same in both cases but latency is better in Edge-based serverless solution. But for heavy workloads, Cloud-based system outperforms the native edge-local alternatives as the later cannot scale beyond the available resources. The high latencies in the cloud system are handled using high scalability and parallelism by processing the requests simultaneously [BMG17].

Serverless computing has an impact on IoT, but running data-intensive tasks in serverless is another interesting insight. The main challenge is to have an effective data communication when running analytics workloads on the serverless platform with tasks in different execution stages via a shared data store. According to Klimovic et al. [KWK+18], ephemeral storage service is needed to support data-intensive analytics on serverless platforms. Ephemeral data is short-lived and by re-running a job’s task data can easily be re-generated. An ephemeral storage system can provide low data durability guarantees.
With the elasticity and resource granularity of serverless computing platforms new research directions arise. Serverless computing is not so feasible for long-lived stateful workloads, though it supports a wide variety of stateless event-driven workloads, with short-lived data, often low-latency requirements, limited-to-no parallelism inside a function and throughput-intensive tasks [KY17]. To support serverless functions, cloud providers handle the burden of allocating resources to users serverless code without prior knowledge of the workload characteristics of the user. Building such systems to meet the elastic application demand is critical. The challenge is to find low-cost allocations that meet the application performance demands with provisioning resources across different dimensions (eg: memory, storage capacity, compute resources and network bandwidth), while keeping high throughput. Ephemeral storage services could be a novel research direction to better serve stateless processing [KWS+18].

### 2.4 Performance of virtualized systems

Virtualization is a key aspect of cloud computing. Virtualization provides scalability, flexibility and effective resource allocation and utilization. According to Huber et al. [HvQHK11], in order to evaluate the performance of virtualized systems, the following research questions arise: i) What is the performance overhead when the execution environment is virtualized? ii) What factors have an impact on the performance of a virtual machine? iii) How does different virtualization platforms performance overhead vary?

To know the performance of virtualized systems, one must know the factors that influence the performance. The factors that influence the performance are grouped into four categories. The first and foremost factor is the type of virtualization. Different virtualization systems have different performance overheads. For example, full virtualization performs better than all other techniques because of hardware support. The second factor is Virtual Machine Monitor(VMM) architecture or hypervisor architecture. For example, better isolation is obtained from a monolithic architecture. The third factor is resource management configuration which in turn depends on CPU scheduling, CPU allocation, memory allocation, number of VMs and resource over commitment. The fourth and last factor that influences the performance is workload profile that is executed on virtualized platforms. Different performance overheads is seen when virtualizing different types of resources.

In the following chapter, we discuss the performance of different virtualization systems.

#### 2.4.1 General

In this section, we discuss the performance overheads of different virtualization techniques and its gaps when compared with native environments. An intense research work has been done on comparing the performance of the virtualized systems and with native systems. We discuss performance, resource usage and power usage overheads of virtualization techniques in clouds. Different benchmarks and performance metrics are considered in order to evaluate the virtualization systems.
According to Selome et al. [TKT18], virtualized systems are tested with different workload types. The workloads are CPU-intensive, memory-bound, network I/O bound, and disk I/O bound with different levels of intensities. The results of virtualization platforms with respect to performance isolation, resource over-commitment, start-up time, and density are also compared. The tests are carried on XEN, KVM, DOCKER, and LXC. XEN and KVM are two hypervisors based virtualization technique. XEN is a para-virtualization implementation where KVM is an open source full virtualization solution that allows VMs to run with unmodified guest OS. LXC and Docker are OS-level virtualization method for running multiple isolated containers on a host using a single Linux kernel.

When running single VM’s/container the performance and resource usage overhead and the results are compared with native environment. CPU usage overhead is almost negligible in all cases. For memory intensive workloads, OS based systems performed better followed by KVM and then XEN. LXC and Docker performed better for disk I/O and network I/O based workloads.

For multi-instance experiments, for resource and power usage overhead, both disk and network I/O exhibited the highest usage by KVM, followed by XEN. VMs provide better isolation and protection against noisy neighbor. In CPU over-commit cases, hypervisors based system performs similar to OS based systems. OS-based systems are more efficient when running start-up time and density tests.

2.4.2 Performance of DBMSs on virtualized systems

Virtualization is used for efficient resource utilization and collocated user isolation in cloud platforms. In DBMS, the underlying virtualization technique has an impact on the performance and isolation mainly in disk I/O. According to research done by Mardan and Kono [MK16] on two virtualization techniques: hypervisor-based virtualization(KVM) and OS-level virtualization(LXC).

The tests are made for disk I/O performance. To test the disk I/O performance without DBMS a flexible I/O benchmark (FIO) is selected. This flexible I/O benchmark produces four workloads: 16KB random read/write and 128KB sequential read/write. For the flexible I/O benchmark, LXC outperformed KVM for all the workloads. To know the performance isolation of KVM and LXC, two VMs/containers are launched to run the sequential write work-load. 30% share of I/O requests is given for one VM/container and the other is given 70%. The I/O bandwidth given to both container and VM are shared gracefully.

To know the disk I/O performance for a DBMS, MySQL server is installed in each VM/Container. To generate the workloads, the Sysbench OLTP benchmark is selected. Two VMs/containers are launched where one VM/container runs MySQL and the other executes sequential write workload of the FIO benchmark. The VM/container running MySQL is given a 30% share of disk I/O and the other is given 70% share. KVM outperforms LXC by 64%. This is because of MySQL issues fsync requests that
keep the file system consistent. The impact of \textit{fsync} is confirmed by proposing three benchmarks: no fsync, low fsync and high fsync. LXC performed better than KVM only for no-fsync. If \textit{fsync} is increased then KVM outperformed LXC. By collocating MySQL with fsync-intensive workloads, the performance of MySQL in containers is improved. LXC outperforms KVM when a normal file system benchmark is executed. KVM (Hypervisor) a better fit than LXC (Container) without violating the performance isolation for hosting DBMS.

There is also a study on the performance of Docker containers with in-memory DBMS (SAP HANA). The research is done by Rehmann and Folkerts, to measure the impact of interference called Noisy Neighbors (NN). The tests are conducted with five OLTP queries with different operations on 2 tables with 100 clients and four OLAP queries work with 38 tables. The maximum number of clients are double to the number of logical cores. The impact of Noisy Neighbors is high in containers compared to VMs \cite{RF18}.

Xavier et al. report due to a NN in containers an overhead of more than 8\% \cite{XNR13}. Interference effect on collocated VMs and containers are investigated by Sharma et al.

From the above-mentioned research work, we came to know that the container outperforms a VM for a normal workload. But on the contrary, VMs outperform containers for database intensive workload. A DBMS running in a hardware-based VM can outperform a containerized DBMS. For relatively small databases shared storage gives better performance compared to dedicated storage.

\subsection*{2.4.3 Hardware-sensitive features and their virtualization}

Multicore platforms consist of both general purpose and accelerator cores. With many cores in a single chip, high throughput and low latency can be achieved. Highly specialized co-processors are often used in database servers \cite{BBHS14}. Processing devices that are used for database operations are multi-core CPU, Graphical Processing Units (GPU), Accelerated Processing Unit (APU), Many Integrated Cores (MIC) and Field-Programmable Gate Array (FPGA) \cite{BBHS14}.

GPUs are designed circuits that perform tasks like rendering videos and high-end graphics games. Development of GPU usage for databases made it encouraging to test them. Nvidia Geforce GPU is used for tests. Nvidia provides Cuda samples that are run on GPU to test the throughput and operational timings.

Jaewook Kim et al. \cite{JKKK18} developed a serverless computing framework based on GPU that uses Nvidia-Docker container. The serverless framework used is an open source framework IronFunctions. IronFunctions is a container-based serverless platform that starts every new service in a container. The main idea of using NVIDIA-Docker is to use GPU in the serverless computing environment. NVIDIA-Docker retrieves information from the CUDA device, volumes, and libraries in the local environment and creates a container with this information. High-performance micro-services are
implemented in a GPU based container. The framework is tested with three scenarios that deals with image processing where the first experiment compare the execution time of CPU and GPU-based services in a serverless computing environment. The second test deals with the execution of a service with deep learning frameworks using remote GPU framework without local GPU against local environment using local GPU. The third test is to compare the execution time of the framework in 1 GBPS and 10 GBPS. There is no GPU and CUDA in the client environment and the server functions are written in python 2.7 and Lua 5.1.

For the first experiment, the functions are written in PyCUDA, SciPy, Pillow, scikit-image and deploy these functions in the IronFunctions framework. PyCUDA functions are executed in GPUs and SciPy, Pillow and sci-kit are run on CPU. The results show that, if the images to be processed are around 10 to 100 the CPU performed better than the GPU based system. The performance is improved by 2.5 to 5 times by using GPU in the serverless environment. When deploying and developing a microservice in serverless computing for image using, using GPU is feasible only if there are more number of images to be processed.

For the second experiment, deep learning frameworks are considered. Two datasets are compared for this frameworks. The two datasets used are MNIST datasets and the other is IRIS flower data sets. The average of 30 times execution time is compared when running on local GPU environment and when run on GPU based serverless environment. For long execution time codes, there is almost no overhead for using remote GPU through serverless computing in terms of response time. For long time workloads Container creation time as well as network latency, computation error in the framework is also negligible.

To run deep learning code in a serverless computing environment, it is important to transfer data from client to server. In deep learning datasets of different sizes are used which vary from KB/s to several GB/s. In the third experiment by using an HTTP REST API deep learning execution code that run in Tensorflow is evaluated. The IronFunctions server is developed on 1 GBPS and 10 GBPS network bandwidth. The performance difference is almost negligible in both 1 GBPS and 10 GBPS network. The performance of file transfer can be greatly improved if the network is configured with a bandwidth of 10 GBPS, but performance or function calls cannot be improved. The larger the data set size is 300MB or more, the bigger the performance improvement.
2.5 Summary

This chapter can be summarized as follows:

- In this Chapter, we discussed types of hardware virtualization techniques. We discussed OS-level virtualization with Docker.

- An introduction of serverless computing, and how is it different from the other cloud services is explained. Examples of applications that uses serverless computing are discussed in this chapter. Vendors and comparisons are discussed, next to additional aspects such as applications with edge computing and proposals for ephemeral storage services.

- This Chapter deals too with details of hardware-sensitive features and its virtualization. We discussed the performance of virtualization in general, for databases, and finally for functions using specialized hardware. A framework that uses a serverless function using Nvidia-Cuda is discussed in detail.

In the next chapter, we introduce our evaluation questions, the prototype that we develop to study them and the experimental settings.
2. Technical Background
3. Prototypical Implementation

In this chapter, we introduce the precise evaluation questions that we seek to answer in our research. The outline for this chapter is as follows:

- We provide several evaluation questions that we aim to address in our study. (Section 3.2)
- A quick listing of the defining characteristics from the execution environment of our tests is discussed in (Section 3.3).
- We describe in detail the benchmarks we used for the tests in (Section 3.4).
- We conclude the whole chapter in (Section 3.5).

3.1 Design - The second step

This chapter documents the second step in the waterfall model that we selected for our research methodology. This stage aims to design the experiments to be conducted. This chapter presents the details of the experimental setup, the tools, and the benchmarks selected.

3.2 Evaluation questions

For the prototypical implementation of the evaluation questions, we have classified them into two categories.

- **Hardware-sensitive features virtualization**
  Development of virtualization is a key aspect in cloud computing. Using containers for database intensive tasks with CPU doesn’t seem to have a positive effect on...
24

3. Prototypical Implementation

DBMS, due to noisy neighbors and limits in sharing the file system. From the research, by considering the current state of art of hardware-sensitive features impact in databases performance, it seems pertinent to consider if there is an overhead from virtualization when using specialized hardware functions. We have selected the following questions:

1. Can hardware-sensitive features be used successfully after virtualization with containers? What is the throughput compared to normal execution? Is there an overhead from the virtualization?

- **Serverless computing**

Serverless computing is a new way of developing micro-service architectures. Every service in serverless computing is developed as service functional units. Every serverless framework at present is CPU based. From the current research state of serverless computing we would like to answer the following research questions that might help research in database systems:

1. Can serverless functions support basic database operations? If so, what is the performance observed when compared to basic execution? Can serverless functions be used to automatically scale-up the processing? What is the throughput comparison using a real-time cloud platform service? Can the cloud emulator performance be replicated in the cloud platform? In addition we provide some sub-questions:

   (a) Can the serverless function be designed to share a common client that reuses connections and resources?

   (b) What is the role of asynchronous clients in providing throughput improvements when compared to other clients?

   (c) What is the throughput when the serverless function is run in a cloud provider, compared to an emulator and to a native execution?

### 3.3 Evaluation environment

#### 3.3.1 Hardware-sensitive features

The initial step before running the samples in native system execution is to install NVIDIA-CUDA in the test system. CUDA is a programming model which is developed by Nvidia for parallel computing tasks. There are some pre-requisites before installing CUDA. The first requirement is to check whether the system has a CUDA capable GPU, with a supported Linux version with GCC compiler installed.

Docker is an open source platform that is used to develop, deploy and run an application. Containers provide an efficient use of system resources. Docker provides a virtual environment to the application by running them in an isolated container. Many containers
can be created on a host machine. Containers are light-weight compared to a hypervisor and are run on the host kernel. By using Docker, with the help of Nvidia-Docker hardware features, like the use of CUDA libraries and drivers can be containerized, making these system resources available to containerized code.

The following configurations are used for the prototypical implementation of hardware sensitive features virtualization:

- **Machine Configuration:**
  - Operating System: Ubuntu 16.01 LTS 64 bit
  - Processor: Intel® Core™ i5 CPU 660 @ 3.33GHz x 4 core
  - Graphics: GeForce GTX 750/PCIe/SSE2
  - Memory: 8GB RAM

- Cuda version: 9.0.61
- Docker version: 17.12.0-ce
- NVIDIA-Docker version: 2.0

### 3.3.2 Native and Cloud emulator evaluation environment

To run the YCSB benchmark in a native system environment, a flask micro web-development framework which is developed using python is used. Flask is highly flexible, lightweight and has a modular design. Flask has a good handling capability of HTTP requests. Flask doesn’t need any tools and libraries in particular. A flask file is developed to connect to Redis-server by creating a client similar to a serverless function. More details regarding the implementation can be seen in Section 5.3.

Serverless functions are written using JavaScript and run in Node.js. Node.js is a JavaScript runtime environment which executes code out of a browser. Node.js has many modules that are used to handle different functionalities. Node.js is an event-driven programming architecture which aims to enhance throughput and scalability. Node.js is a single-threaded asynchronous architecture that guarantees scalability without threading. Node.js is used to build scalable servers, and by using the callback function the status of the task is monitored. By using Redis module in node.js, the function is developed to create a Redis-client in the Redis-server host address to store the data.

Redis is a fast and easy to use in-memory data store which is used as a database or as a cache. Redis is treated as a data structure as the key contains hashes, strings, sets, and lists. Redis doesn’t have any concurrency problems as it is single threaded. Redis is persistent as the dataset snapshots are stored frequently, however it can also be configured to run only in memory. A client/server protocol is needed to interact with Redis. Redis has clients written in many scripting languages. For the implementation,
python client redis–py is used for native execution and node_redis client is selected for implementing in a serverless environment. Node_redis supports all the Redis commands and it aims for high performance. The function connects to Redis and performs the basic database operations by loading the data from YCSB benchmark.

To run serverless Node.js function a cloud emulator is required. An Emulator is a nodejs application that implements cloud functions. A cloud emulator is installed using npm install command. Before deploying the serverless functions in cloud platforms, the emulator provides an option to deploy, debug and run the cloud functions in the local machine. If the deploying of a function is successful then the function can be deployed in cloud providers. With the help of an emulator, the cost for running a function in the cloud platform is reduced. Installation of an emulator is verified using the functions start command which starts the emulator.

The emulator has two configuration parameters MaxIdle and IdlePruneInterval. A maxIdle time is defined as a connection that can remain in a connection pool but is unused before being discarded. If there are 5 Connections in the pool and has no activity, after maxIdleTime has passed, all the connections will be expired and new connections begin. IdlePruneInterval is used to automatically close the connection after being idle for a particular interval of time. By changing the values of these two configurations the performance of the functions deployed in an emulator can be varied. More information about the cloud emulator is found in the Google cloud official documentation.

To implement serverless features the following system configuration and versions are used

- Machine Configuration:
  - Operating System: Ubuntu 16.01 LTS 64 bit
  - Processor: Intel® Core™ i5 CPU 660 @ 3.33GHz x 4 core
  - Graphics: GeForce GTX 750/PCIe/SSE2
  - Memory: 8GB RAM

- Redis version: 4.0.1

- Python version: 2.7

- Node.js version: >= 6.11.1

- Java version: 1.8.0_181

- Flask version: 0.12.4
3.3.3 Cloud platform

To implement the serverless function in a real-time cloud service provider, the Google Cloud platform was selected. It is a cloud computing service that provides compute services like Infrastructure as a Service, Platform as a Service, and also Function as a Service. It also supports data analytics, data storage, Networking, IoT and machine learning services. Google cloud platform is available in 17 regions with 52 available zones. Users can deploy the required cloud resources in any region. In a region, there are different availability zones. Most of the regions have three or more availability zones. The best practice is to select the closest region available to reduce the latency.

As Redis uses a client-server protocol to communicate, two virtual machine (VM) instances and a cloud function are created in the Europe region. An instance is a virtual machine which has processor and memory and runs an operating system. All the instances created in Google cloud are hosted on its own infrastructure. For each instance, the number of virtual CPUs and memory can be selected. A machine type feature is provided to define the resources that are available to an instance. The resource information includes memory size, virtual CPU (vCPU), and persistent disk capability. Depending on the tasks that are performed in the instance the machine is selected.

In the cloud platform, in order to connect from one instance to another instance there has to be common firewall rules. Firewall rules are used to allow and deny the traffic to and from the instances. Firewall rules provide protection and traffic control on instances. Firewall rules need to be configured in order to connect from one instance to another instance in the same Virtual Private Cloud (VPC). More information regarding the Google cloud platform is available in the official documentation.

From two created instances, one instance is treated as a client which connects to the other instance where Redis-server is running. In the client instance, Java default JDK, maven, Nodejs, and Redis-tools are installed to make a successful build of ycsb workloads. Redis-server is installed in the server instance.

The instance configurations and installed software versions in both the VMS are:

- Virtual Machine Configuration of both instances:
  - Operating System: Ubuntu 16.01 LTS 64 bit
  - Machine type: n1-standard-4 (4 vCPUs, 15 GB memory)
  - CPU platform: Unknown CPU platform (Selects randomly from available CPUs when an instance is started)
  - Zone: europe-west-1b
  - Graphics: NVIDIA Tesla K80 (Only in Redis-server instance)

- Redis version: 3.2.6
- Python version: 2.7
• Node.js version: >= 6.11.1
• Java version: 1.8.0_181
• Flask version: 1.0.2

3.4 Datasets

• NVIDIA-Cuda samples

To test the performance of GPU, we have selected default NVIDIA-CUDA samples that are provided when CUDA is installed. CUDA is a programming model and a parallel computing platform invented by NVIDIA. Computing performance is increased by exploiting the power of Graphics Processing Units (GPUs). GPUs that use CUDA have hundreds of cores that simultaneously run thousands of computing threads. To test these samples, CUDA toolkit is installed. A detailed explanation of CUDA installation with pre-installation requirements and a step-by-step procedure is specified in the official CUDA toolkit documentation.

• Yahoo! Cloud Serving Benchmark

To evaluate the performance of the serverless functions by loading and running the data for basic database operations for different workload proportions we consider Yahoo! Cloud Serving Benchmark.

In recent years, there is a huge development of data serving systems in the cloud. Open source systems include Cassandra, HBase, Voldemort, and others. Some systems are offered only as cloud services, either directly in the case of Amazon SimpleDB and Microsoft Azure SQL Services, or as part of a programming environment like Google’s AppEngine or Yahoo!’s YQL. These systems don’t support ACID transactions but address cloud OLTP applications. The emerging cloud serving systems and the applications that they are proposed for lack the performance comparisons. It is hard to predict the relationship between systems and the workloads that are best suited for it. To overcome this problem, a Yahoo Cloud Serving Benchmark framework is proposed, with an idea of comparing the performance of cloud data serving systems. YCSB provides a provision to test them against one another on a common base and provides a better solution to select a database. YCSB is used to evaluate the performance of different key value stores and cloud serving stores by developing a framework and a set of common workloads. [CST+10]

YCSB consists of a Client as a workload generator and a YCSB core package which has standard workloads which act as a benchmark for cloud systems. In the workloads, the data loaded into the database during a load phase and database operations are performed on dataset during run phase is described in the workloads. Each workload has Read, Scan, Update and Insert proportions.

YCSB benchmark has six workloads in the core package. These six workloads have a similar dataset. The workload proportions are:
3.5 Summary

The chapter is summarized as:

- This chapter focuses on the evaluation questions that we would like to answer from our research.

- We also detailed about the experimental setup that is used in our work. The containerization tool used to implement hardware-sensitive features, and different cloud platforms along with Redis, Node.js to implement serverless features are explained in detail.

- The samples and the benchmarks used for the tests are also presented.

In the next chapter, we present the implementation of our first evaluation question, containerization of hardware-sensitive features, running the sample tests and evaluate the results, provide the summary and discuss them in detail.
4. Hardware sensitive features

We outline this chapter as follows:

- We establish the evaluation questions that motivate this chapter. (Section 4.2)
- We answer the evaluation questions regarding experimental analysis and results. (Section 4.3 and Section 5.4)
- To conclude we summarize the work in this chapter. (Section 5.5)

4.1 Implementation - The third step

This is the third step of our research methodology based on the waterfall model. This stage aims in implementing the experiments from the design phase. This chapter presents the execution of the first evaluation question.

4.2 Evaluation Questions

As discussed in Chapter 3, the hardware sensitive features have an impact in the database performance. Unlike a virtual machine by containerizing the hardware features, all the applications running in containers are able to use the system resources by sharing the same host kernel. This feature of containers gave an insight of containerizing a GPU and run sample tests to check the overheads when compared to normal GPU execution.

1. Can hardware-sensitive features be used successfully after virtualization with containers? What is the throughput compared to normal execution? Is there an overhead from the virtualization?

   (a) How are the hardware sensitive features based tests RUN on a native environment and in a virtualized environment?

   (b) What are the tests that are selected to compare the performance overheads?
4.3 Implementation

In this section, we discuss the implementation of how the samples are run in native system and also a step-by-step procedure of hardware-features virtualization and running tests.

4.3.1 Native system execution

To run the tests in the native system, CUDA samples are selected. The samples consist of different types of references like a simple reference, utilities references, and also imaging, graphical, and simulation references. Simple references are used to understand the concepts of CUDA and its runtime APIs. Utilities reference samples are used to measure the CPU/GPU bandwidth. As the name suggests, imaging reference has samples that deals with imaging and data analytics, financial reference samples deals with the parallel algorithms in financial computing tasks.

Before selecting the tests for comparison, all the test samples in the samples sub-folder of the NVIDIA installation folder need to be executed. To achieve this, a shell script is written in such a way that, all the tests in the samples folder are built with the `make` command first and then all the tests are executed by saving the output to a text file. From the results of all sample tests, one test from simple reference, utilities reference, finance reference is selected to compare the output with the container based execution.

4.3.2 Virtualization of hardware-sensitive features

Docker is used to containerize the hardware features. As discussed earlier, docker is a containerization tool used to develop applications in isolated environments. After successful installation of docker in our local machine, the main task is to develop a docker-file. A docker-file is used to start a container from a base docker image. A docker file is build using `nvidia-docker build` command. When the build command is executed the following steps will start:

- step:1) Docker pulls the image from the docker hub and starts a container from `NVIDIA/CUDA: 9.0` base image. Make sure the CUDA versions running in the host system and in the container are same.

- step:2) The next step is to install CUDA toolkit. In this stage, the sample tests that are selected to compare the performance are copied to the container from the host machine by using docker `COPY` command.

- step:3) After adding the tests into the container, by using `make` command the tests are build and ready to be executed.

- The final step in docker-file is to copy the shell scripted file which runs all the executable files(.sh files) in the samples folder in the container and save them to a text file.

The tests that are performed, and the results obtained are plotted and discussed in the next section.
4.4 Evaluation

This is the fourth and the final stage of our research methodology in the waterfall model for the first evaluation question. In this section, we present the results of Cuda sample tests for two executions.

The tests are selected to work with the CUDA concepts like asynchronous data transfers, CUDA streams and events, and also for computational tasks.

4.4.1 asyncAPI

It is a test sample of simple reference. AsyncAPI test is the test made to determine the overlapped execution of CUDA streams in CPU and in a GPU. The test provides the information of time taken to execute test using a GPU and also the time that a CPU spent for CUDA calls.

![Figure 4.1: Comparison between normal execution and virtualized execution of hardware sensitive features for asyncAPI](image)

From Figure 4.1 it is evident that the GPU running in native execution spent more time to execute the test when compared to the virtualized GPU execution. Containerization has an advantage when compared with the native execution, but the difference in time to execute the test is almost negligible. In order to understand better, the time that the CPU spent for CUDA calls was considered. It is the same in both the cases. But the number of cycles that the CPU executed while waiting for GPU to finish is higher in native execution compared to containerized execution.
4.4.2 SimpleMultiCopy

This test sample belongs to simple reference in CUDA samples. This test is selected because, it covers two aspects of CUDA concepts which include CUDA streams and events, asynchronous data transfer. This test uses CUDA streams to observer the performance of GPU by the overlapping of kernel execution with data copies to and from the device. A host system has one or more CPUs and a device is a GPU that runs concurrent threads. The difference between host and device is based on threading resources, threads, and RAM. The threads in a CPU are treated as a heavyweight entity. In the GPUs, the threads are very light-weighted entities. The data needs to be transferred from host to device in order to use CUDA by using the PCI-e bus. The data to be transferred should always be placed in the device rather than on the host.

![Figure 4.2: Comparison between normal execution and virtualized execution of hardware sensitive features for SimpleMultiCopy](image)

From the Figure 4.2, the time taken by virtualized execution is less than native execution. The difference in the measured time is almost negligible, which suggests that virtualization of hardware resources has no effect on the performance of hardware-sensitive feature when compared to native performance.

4.4.3 Bandwidth Test

This test is a sample from the utilities reference. Bandwidth is generally defined as the rate at which data is transferred. Bandwidth is the key factor to determine the performance. This test is used to measure the memory bandwidth among the CPU and
4.4. Evaluation

GPU and between GPU addresses. This test is similar to simplemulticopy test, but the difference is, this test records the bandwidth when data with a transfer size of 33554432 bytes is copied from host to device, device to host and device to device.

![Comparison between normal execution and virtualized execution of hardware sensitive features for Bandwidth test](image)

Figure 4.3: Comparison between normal execution and virtualized execution of hardware sensitive features for Bandwidth test

From Figure 4.3, the data transferred from device to host and vice-versa has higher bandwidth in containerized execution, but in case of memory copy from device to device the native execution has better bandwidth than the other execution. The drop in the throughput in the containerized execution is because of the kernel. When a kernel writes or reads data from device memory, it affects the host to device transfers that are happening concurrently. The bandwidth varies with a particular amount of an overhead below 256KB of data size. The effect of changing overheads reduces if transfer size increase beyond 256KB in the device to host and vice-versa.

4.4.4 Blackscholes

This model is used to estimate the cost of European finance markets. This sample focuses on providing the performance of GPU depending on the options for computing task. The kernel for blackscholes is developed by Nvidia. BlackScholes has a call option and a put option. An option is a right to either buy or sell a product depending on the particular conditions over a period of time. This test allocates CPU memory, GPU memory for options and generates input data in CPU memory and then copies the input data to GPU memory.
4. Hardware sensitive features

Figure 4.4: Comparison between normal execution and containerized execution of hardware sensitive features for BlackScholes test

From Figure 4.4, the effective bandwidth when an option size of 8000000 with 512 kernels is obtained. The native execution performed better than container execution with a negligible difference in throughput. The performance lack in Containers is due to the kernel sharing feature of the container. The GPU runtime is little higher in container execution. The memory of CPU and GPU are released after the tests are executed.

4.5 Summary

This chapter is summarized as follows:

- In this chapter, we provided the results of hardware-sensitive features. The tests involved with the calculation of bandwidth and the measured timings. The tests are conducted for asynchronous data transfer and utilization of CUDA streams and events.

- The most important outcome is that if the tests are hardware-sensitive based, there is no difference in performance overheads when executed natively or in containers. If the tests are based on kernels, there is a drop in the performance of hardware-sensitive features in containers because of sharing a common kernel feature of containers. Though the performance drop is almost negligible.

- The performance of containerized execution is good because the containers are light-weight in nature and has less startup time, which makes the execution faster.
As the throughput is almost the same in both cases, the next insight would be to implement this in the GPU based databases to utilize the better performance from containerization. In addition it would be important to study how problems of noisy neighbors and sharing underlying file systems could be alleviated for using GPU databases with containers.

In the next chapter, we discuss the second evaluation question.
5. Serverless Computing for databases

We outline this chapter as follows:

- We establish the evaluation questions that motivate this chapter. (Section 5.2)
- We answer the evaluation questions regarding experimental analysis and results. (Section 5.3)
- We collect the findings of this chapter in a list of best practices. (Section 5.4)
- To conclude we summarize the work in this chapter. (Section 5.5)

5.1 Implementation - The third step

This is the third step of our research methodology from the waterfall model. This chapter presents the execution and the results for the second evaluation question.

5.2 Evaluation Questions

As discussed in chapter-3, the serverless function is implemented in both the native system and the Google cloud emulator.

2. Can serverless functions support basic database operations? If so, what is the performance observed when compared to basic execution? Can serverless functions be used to automatically scale-up the processing? What is the throughput comparison using a real-time cloud platform service? Can the cloud emulator performance be replicated in the cloud platform? In addition we provide some sub-questions:
(a) Can the serverless function be designed to share a common client that reuses connections and resources?
(b) What is the role of asynchronous clients in providing throughput improvements when compared to other clients?
(c) What is the throughput when the serverless function is run in a cloud provider, compared to an emulator and to a native execution?

5.3 Implementation

![Diagram of implementation process](image)

Figure 5.1: Implementation
5.3.1 Native System Execution

To run the YCSB benchmark in Redis, a flask file is developed. The flask file acts as a middleman that connects YCSB benchmarks and the Redis. The process of executing YCSB benchmark using a flask file is discussed in detail below:

- The initial step in developing a flask file is to import Flask, usekwargs, fields, validate, parser and redis. After importing necessary packages, a connection to Redis-server instance needs to be established. Redis-server runs in 'localhost' address at default port '6379'. Once the connection is created, the code is written to upload and retrieve data from Redis. The code we developed is present in Section 7.0.2.

- For every Redis key, ten field values are stored. This is done with the help of Redis-py, a Redis client which acts as a python interface to Redis-key value store. By using the Redis hmset, hgetall, hdel, and hscan commands the basic database operations like insert, read, scan, and update are executed in Redis.

- Now the Redis-server is started, and then the middleman is started running. It is recommended to have a Redis-server running before the flask file started running. To check whether the middleman inserts and reads the values from Redis, a small test of sending a JSON payload from the curl request is done. The key and the field values for a User or Table are sent as JSON payload to store in Redis. Depending on the request method as PUT or GET from the curl request the database operations are executed. A PUT request is always executed first before GET request as the data needs to be stored in Redis.

- The data stored in Redis is accessed with Redis-cli as shown in Figure 5.2 or by GET request from curl. Redis-cli is a Redis client that connects to Redis-server with localhost address(127.0.0.1). Once the Redis-cli is connected to Redis-server, KEYS * command is run to display the stored keys in Redis. The values for particular keys are obtained by running basic Redis hgetall command. After successfully storing data in Redis from curl request, the next step is to start running the YCSB tests in Redis with necessary steps.

- In the YCSB, in a redis sub-folder the Java file is modified such that the file starts a HttpURLconnection with a request type and the request property. The URL runs in an HTTP endpoint which is obtained after running the middleman. An example URL for insert operation looks like this "http://localhost:5000/insert?table=User&key="+key;" is added in the YCSB benchmark.
In the YCSB, the Java file is developed depending on the request method. For insert and update operations in Redis, PUT request method is used. For read and scan operations, GET request is specified. The pom file in YCSB and in Redis folder are added with necessary dependencies for successful maven build operation of Redis-binding. Once the maven build is successful, YCSB tests are run in Redis-server by providing redis connection parameters like workload to Load and Run, redis.host, redis.port. The Redis.host address is the IP address of the machine where redis-server is running. It is 127.0.0.1 in this case and port address is 6379 the default port where redis runs.

By Loading data and Running the workloads, the output for respective operations are stored in text file to measure the average throughput and latency. To check whether the tests are successful, we use the redis-cli command and KEYS * command as mentioned earlier to display the key values that are stored in Redis from YCSB workload.

From six different workloads of the YCSB benchmark, five workloads are selected which deal with insert, read, update and scan are loaded and the tests are run. This is how different YCSB workloads are run in the Redis for native execution environment and the outputs are stored in a text file. The performance of the native execution is discussed in detail in Section 5.4

5.3.2 Cloud Emulator execution

Running the YCSB benchmark in the cloud emulator is different from the native execution. Unlike native execution which uses flask file as a middleman to connect Redis and YCSB benchmark, emulator execution runs a Node.js script. The step-by-step procedure of how the emulator execution is done is seen discussed below.

- The initial step is to select HTTP and Redis modules from the node_modules. A Redis client is created in the host address where Redis-server is running. As the Redis-server and client both are running on the same local machine, the host address is generally a 'localhost' or '127.0.0.1'. The port address on which Redis is running also needs to be specified. The default port where Redis runs is '6379'.

- After successful client creation, the function is developed in a way that, depending on the request method either POST or GET, the function reacts accordingly and processes the requests. For the POST request, the function is developed to set the values of ten fields for a single key in Redis. If the workload inserts or updates the values, then POST request processes it. For the GET request, the values corresponding to a particular key are read from the Redis. The GET request processes the read and scan operations in Redis.

- Once the function is developed, it is deployed with the functions deploy command of the emulator by specifying the trigger type (-trigger-http) used to invoke the
function. If the deployed function is error free, the emulator provides an HTTP endpoint where the serverless function is running. This HTTP endpoint provided by the emulator is added in the URL string of RedisClient.Java file present in the YCSB benchmark. The HTTP endpoint makes sure the test is run in the cloud environment rather than a local machine.

- Once the HTTP endpoint is updated in the YCSB, then maven build is done to make sure the RedisClient.Java is error free. Then the same process of running different workloads with the connection parameters are specified to load the data and run the tests. The throughput and latency obtained for all the workloads are saved. The uploaded keys can be seen in Figure 5.3.

If the performance of the function deployed in the emulator needs to be altered, then the emulator configuration can be changed and the tests can be re-run to get a better performance. In our work, the emulator parameters are changed to get the better results for the serverless function. The detailed discussion on the performance of serverless function execution with default and the changed emulator configuration is discussed in Section 5.4.

5.3.3 Cloud platform execution

In the cloud platform, the same cloud function developed during emulator execution is used. But the process of running the tests are quite different. In the cloud platform, the host address where the redis-server runs is different from the emulator.

- In the cloud platform console, a project is created first. After creating a project, from the compute engine section two virtual machines are created in same region. One instance runs Redis-server and the other instance runs the YCSB workloads. Every instance is provided with SSH option to login into the VM instances. All the necessary software are installed in the instances, information regarding installed tools and their versions is provided in Section 3.3.3.

- From the console, using the cloud function section a new cloud function is created. It is suggested to create the function in the same region where VM instances are created. After uploading the function, an HTTP endpoint is obtained similar to emulator execution. It is important to use the external IP address of the Redis-server instance. If the internal IP address is used, the YCSB cannot run the workloads.

- To connect two Redis instances in a client/server protocol model, the redis.conf file needs to be changed. The bind address must be 0.0.0.0 in-order to accept the connection from any client instance.

- The obtained endpoint is then updated in the RedisClient.Java file present in the YCSB instance. After updating the Java file and the pom file, the workloads are
run from YCSB instance that creates a client in the Redis-server instance and performs the insert, read, scan and update operations. With the help of view Logs option in cloud functions, the status of the process can be monitored immediately.

- By connecting from client instance to server instance using `redis-cli -h <IP address of Redis-server>`, we can verify whether the KEYS are loaded into Redis or not as shown in Figure 5.3.

This is the process of how a serverless function is executed in the Google cloud platform. The throughput and latency for each test are saved to a text file for comparison against other executions. In the next section, the outcomes of all the executions are discussed along with the reasons for their performance.
5.4. Evaluation

This section is the fourth and the final section of our research methodology and presents the results for second evaluation question.

In this section, the comparison of different workloads in different executions are discussed and compared.

- The native execution occurs as explained for traditional implementation.
- Next, We report the execution of a serverless implementation using a local cloud emulator, with a default configuration.
- We report a similar execution but with a change in configuration consisting of maxIdle and IdlePruneInterval. By reducing these times, we close the connections early and start a new connection as soon as the old connections are killed.
- Finally, we evaluate Google cloud platform without any change in the configuration, since it is not possible.
- We also evaluate, the native execution in Google compute instance, which shows the best performance overall.

The comparison is done by considering the throughput and latency for Load and run the data from YCSB.

5.4.1 YCSB data Load

1. Throughput comparison

In order to test the performance, the YCSB data is first loaded from the workloads. This uses the PUT or POST request method and uploads the values in Redis. When the tests are loaded, insert operation is performed on the Redis. The performance for different workloads are discussed below:

From Figure 5.4, Out of all the executions, Load operation of YCSB produced better throughput in the emulator with the default configuration. Using a Node.js
environment is faster than using a python file to update the values into the Redis. The emulator performed better because of the event-driven architecture of Node.js which makes concurrent requests by using a single thread. This feature of Node.js helped in making the emulator perform better. But when the emulator configuration is changed, the throughput is decreased. This is because of reducing the idle time of the clients which reduces the concurrent calls from the connection pool to insert the data into the Redis.

The configuration change is, the change in the parameter values of the cloud emulator to impact the function running in it. By running the `functions config list` command, a list of parameters that can be changes will be displayed. In our research, we considered maxIdle time and IdlePruneInterval which deals with clients in connection pool. The default values provided by the emulator is so high, we changed these configurations to least possible value to see the difference in the output. But, we found the better results are obtained when these these both parameters are set to 500. This is done using `functions config set maxIdle 500` command, similarly for IdlePruneInterval.

![Figure 5.4: Throughput of YCSB Load operation for all executions](image)

After analyzing the results from the emulator, the general belief is to see a similar or a narrow variation of performance in the cloud platform. The reason behind this belief is the implementation of the same serverless function in the cloud emulator and cloud platform provided by Google. But the results seem to be a quite different than expected. The throughput in the cloud platform is very low.
When the native execution, i.e., a flask file is used for data ingestion in a Google cloud instance, the performance is high compared to all other executions. The latency and throughput results are better than the native execution in local machine. This approach uses client/server model in a single compute instance machine, which is the reason for better performance.

![Figure 5.5: Throughput of YCSB Load operation using serverless function in Google cloud](image)

This is because the number of operations performed on the Redis-server is less for a given time. From Figure 5.5, to execute a few hundreds of requests, it takes couple of minutes to process them. The throughput depends on the host resources like CPU, network, and the operations performed. The operations performed by the database is very low in this case. This problem is not seen in the emulator because, both the emulator and the Redis-server are on the same machine which makes the execution faster. There is no problem with the networking in case of emulator execution which processes more operations.

2. Latency comparison

When comparing all the executions, the Latency is less in the emulator with the changed configuration for all the workloads. Average latency between native and emulator with default configuration differs narrowly.

In the native execution, for workload c which is 100 percent read has high average latency when compared to the other workloads. In the emulator with default configuration, for the workload b, with 95 percent read proportion, and 5 percent upload proportion the average latency is high.

From Figure 5.6, the latency in the cloud platform in very high compared to the emulator execution. This is because of using two VM instances the time taken to
complete the operation is high. To insert values into Redis, for each key that is being inserted, the type of request is verified and then function starts executing again. The process of using a switch case condition to check the type of request for every key from the YCSB takes time to finish. A way to improve the latency is to reduce the time taken for the function execution by having a high-speed network connection between the instances.

Figure 5.7 is the snapshot of the log that is used to track the process during the execution of a serverless function. In the Log file, it is evident that the function starts for every request to insert the values. The function takes different time to finish executions. Sometimes the function finishes within 3ms but sometimes it takes around 70ms to finish the execution. So the difference in the time taken to execute the function for each request is the reason for the high latency in the cloud platform.

To reduce the average latency and improve the throughput in the cloud platform, the tests need to be performed on a single VM instance instead of two. But the problem with this approach is the basic concept of a client/server model of Redis is not achieved. The change in configuration of the emulator has improved the average latency by making it better compared to all the executions.
5.4.2 YCSB data run

1. Throughput comparison

In this section, the performance of serverless in different executions is analyzed when the YCSB workloads are run.

From Figure 5.8 the native execution outperformed all the other executions. To retrieve the data from Redis, Flask performed better compared to the Node.js. With the default configuration of the emulator, the throughput is very low. The reason for this is the redis client connection. For every GET request, redis creates a client and then reads the field values from the Redis. By default, the maxIdle time for connections is huge in the emulator. It takes long time to close the previous unused connections and create new connections. This is the reason for having a low throughput when a serverless function is run in an emulator with default configuration.

But by changing the configuration of the emulator, the time to close the unused connections is reduced and the interval time to prune (close) the unused connections are reduced to 500. After changing the configuration, new connections are closed and started frequently when compared to the default execution. The best configuration that produced better throughput compared to default execution is by setting maxIdle and IdlePruneInterval to 500. From Figure 5.8 it is clear that the throughput increased drastically for all the workloads with the changed emulator configuration. But the throughput from changed configuration is never close or higher than the native execution.

For YCSB run operation using the native execution flask file in cloud platform, the throughput and latency are high. The cloud platform execution using flask file has outperformed all other executions. A next insight would be to use flask file to run the YCSB tests with two compute instances could produce better output compared to what we have seen from serverless function.

The throughput of the cloud platform is the lowest of all the executions. The number of operations performed between two instances is low. The performance of Redis is bound to memory or network. For each request, the function starts executing which is the drawback to perform more operations in a particular amount.
of time. Gcloud doesn’t have any configuration properties to change and make the throughput better.

2. Latency comparison

In general, the native execution has low average latency when compared to other executions. The latency is quite interesting in case of update-intensive workloads in all executions. The average latency for upload operation is low than other database operations for all the executions. In the cloud platform, all the operations except update has very high average latency. The average latency for read, insert and scan operation in a cloud platform is very high.

The change in configuration of the emulator doesn’t have a significant effect on the average latency as it had on the throughput discussed above. The average latency varies in a range of hundreds in default and changed emulator configuration. This suggests that change in configuration doesn’t have any positive effects on time taken to complete a task as far as the average latency is concerned.

From Figure 5.9, the workload with 95%-5% read-update proportion the update has less average latency in the cloud platform compared to an emulator. From v2 which is a 50-50 read-update proportion, the average latency is low in the cloud platform compared to all the executions. No exact reason for this behavior is known and could serve as a future aspect to research, but the outcome is, the
update has less average latency in all the executions, and especially in case of cloud platform the low average latency is encouraging.

5.5 Summary

In this chapter, we discussed the implementation and the performance of YCSB workloads for different executions. First, we discussed the implementation of YCSB benchmark using a flask file. With the similar logic used in the flask file, a serverless function is developed using a node_client. This client is a python interface to the Redis key-value store.

- From the tests and results, by testing YCSB benchmark using python programming and Node.js environment gave a good outcome when executed in the host system. It is interesting to run the YCSB benchmarks in two interconnected local systems to check the performance and compare it with the cloud platform execution. This would give more insight of network-bound feature of Redis.
• The cloud emulator with default configuration and the cloud platform throughput for YCSB run is low. The cloud emulator execution performed better only after the configuration changes (maxIdle and IdlePruneInterval) which cannot be done in the Google cloud platform. It is interesting to work to check if gcloud provides such configurations that can impact the throughput in the cloud platform.

• In the emulator execution, changing the configuration (maxIdle and IdlePruneInterval) has drastically improved the throughput of YCSB run. This suggests the importance of configuration parameters that has an influence on the performance of cloud emulator.

• In the localhost execution to LOAD the data into Redis, it is feasible to use serverless function and to RUN the tests a python file is needed to achieve a better performance by considering throughput and latency. By using the cloud emulator with changed configuration, we get a performance which is close to native execution.

In the next chapter, we conclude our work, give threats to the validity of our evaluations and propose future work.
5. Serverless Computing for databases
6. Conclusion and Future Work

This chapter is discussed as follows:

- We conclude our work by focusing on the important aspects of our research, reviewing our findings and summarizing our approach (Section 6.1).
- We disclose possible threats to the validity of our study (Section 6.2).
- Finally, we highlight particular areas in this domain, where future work can be done (Section 6.3).

6.1 Summary

The growing amount of data in today’s world needs a better way of handling. The option of handling the data using traditional resources doesn’t make the database systems fast. In order to handle the data fast in a database, there are two general choices scaling up and scaling out. These options are catered for in novel ways by cloud providers, with scaling up being possible with container technologies and the renting of GPUs and specialized processors, and scaling out being made possible with serverless functions.

Hardware-sensitive features need tuned-algorithms that brings the better out of them, but managing large scale distributed systems to be able to use hardware sensitive features efficiently can be difficult. For this container technologies seem promising.

On the other hand, serverless functions use features of event-driven architecture and non-blocking I/O, which does not block program execution under I/O-heavy workloads and maximizes the utilization of a single CPU and computer memory, making services fast and productive. However the logic of these frameworks offloads the scheduling of tasks from the database or OS into the cluster manager.
The need for database systems to be fast and efficient both in their processing and in their management creates interest in studying the applications of these techniques for database tasks. A summary of our research work is provided below:

- The steps provided in the waterfall model are used to produce useful outcomes. This model helps in making the research reproducible.

- The aims of this research are: first, to analyze the steps needed to virtualize hardware-sensitive features and evaluate their performance compared to basic execution; second, to develop a serverless function as part of a database benchmark, evaluate the tasks of data ingestion, data run using database operations, and evaluate how that could be made to work efficiently in cloud platforms.

- To carry out the research on hardware-sensitive features CUDA and Docker containers are used. CUDA is a GPU programming model developed by NVIDIA. Docker is a containerization tool used for OS-level virtualization. The light-weight, easy to build, ship and deploy feature of Docker made it an obvious choice to work in the research.

- For research on serverless functions Redis, a key-value store is used. Redis has flexible modules and clients that improves its potential to more than a key-value storage. We used Redis module to connect to Redis and an HTTP module for HTTP requests. We used a python client, redis-py a python interface to the key-value store. We also worked with asynchronous clients ioredis, promise and when.promise to improve the performance of serverless functions.

- For the two evaluation areas, different tests are implemented. For the hardware-sensitive features, the samples provided by CUDA are tested containerizing them. For the serverless function implementation, the tasks are categorized as, data ingestion to load data, and data run.

- In the hardware-sensitive features, the samples are run on a native system. Then a container is created using a docker file which executes the same samples. The tests are selected based on the impact they have on GPU. It has been seen that, there is no difference in the performance of the GPU when running natively and in a container.

- For the serverless function, first we discuss the performance of a serverless function in cloud emulator and cloud platform and compare it with the native execution using flask for YCSB workload data ingestion into Redis. The results show that the cloud emulator performed better compared to all other executions. The time taken by a function to finish executing is high in cloud platform compared to the other executions. This is the reason for the negative performance of Google cloud platform.
• For the data query, the change in emulator configuration (maxIdle and IdlePruneInterval) has improved the throughput for all the workloads by a large ratio. Though the throughput has increased drastically it is not close enough to the native execution using flask file. The flask file execution has the best performance compared to all other execution. The Google cloud platform has the least throughput out of all. But the Google cloud platform has low latency for the workloads that deal with UPDATES.

From our research, we conclude that serverless functions can be used for data ingestion as the performance is high when compared to all other executions. It could be more beneficial than what we report with more auto-scaling available. For data querying, the serverless function performed better only after changing the maxIdle and IdlePruneInterval. Even though, by changing the emulator parameters, the serverless function still lags behind the native execution. For Google cloud platform it is tough to increase the performance of our serverless prototype, as for each request, the function starts executing from scratch, which adds latency. In the cloud platform, to perform a few hundreds of operation it takes around 4-5 minutes which in-turn results in reduced throughput. This results state that the implementation of serverless functions using two instances in a Google cloud platform doesn’t have much positive outcomes.

6.2 Threats to validity

This section deals with the threats to validity of our results and difficulties faced in this work.

• CUDA allows to develop new samples from the basic samples they provided. We used default samples to run the tests for hardware-sensitive features and didn’t explore or modify any feature in the samples. By developing new tests to have more impact on GPU performance, our results could have been better.

• The serverless function implementation is done using node v6, but using the latest version might have made the results even better.

• Different versions of Redis used in cloud emulator and in cloud platform may have an influence in the performance of the serverless function.

• The system configuration used for the cloud VM instances can affect the performance. In the Google cloud, we opted to make the cloud provider allocate the available CPU for the machine. Each time the machine is restarted, the CPU platform changes, which had an impact on the results.

• By using different kinds of standard datasets (e.g. other scale factors for YCSB) and workloads (e.g. more comprehensive applications than YCSB), the overall results might have given better insights about using serverless functions for database tasks and containerizing hardware features.
6.3 Future work

In this work, we tried to improve the performance of database operations using serverless functions and provided some outcomes that we hope could help the database community.

The serverless function has not been used for database applications till now, this research would serve as a starting step for future research. Though the outcomes from our research require some further evaluation, there are many open questions for the future work in this field and areas where better results can be achieved with improved research. Using latest versions of node, Redis and different Redis modules will definitely improve the performance of the serverless function. The workloads used for the test are mostly Read intensive, digging more on creating new workloads with different work proportions would give a better insight into the serverless performance for other database operations. Redis offers more modules that we didn’t include in our work due to time limitations. More modules, including some publish-subscribe applications, might introduce different insights about performance.

We suggest that some future work should improve the throughput and latency for data loading in the cloud platform using scripting, such that the performance matches the normal load execution. From the latency of data query, the UPDATE latency is very low in the cloud platform compared to all the execution latencies. Further study on this could be a valuable insight on how serverless functions behave for UPDATES. More study is required on why the emulator performance is not replicated on the cloud platform.

Testing the serverless function on other cloud platforms with scripts written in different programming languages can provide positive insights. In our research, we worked only with one cloud platform, the next step would be to work on popular cloud platforms and compare their performance.

Serverless functions can also be tested using the default databases provided by the cloud platforms. This may have a better performance rather than the approach that is used in our research of having two instances and making one of them a server and other as a client.

We believe that the offering of serverless functions with some partially stateful aspects, like shared clients, could plausibly be offered in future versions of serverless frameworks, such development could make a big impact on the readiness of the technology to be used with databases.

From our research, for GPU intensive tasks, there is no drop in performance compared to normal execution. After finding that single GPU processes can be used from containers with little overhead from virtualization, it becomes relevant to design tests to evaluate multiple GPU containers and the impact of resource sharing. The next idea would be to run and analyze the performance of GPU based databases using container technology over virtual machines (VMs).

Finally, we consider that future work depends on how the serverless function can be developed in such a way it benefits the database operations. As the implementation
of serverless functions for databases is still in the early stages, proposing new ways of testing them in different cloud platforms, using the default databases provided by the cloud platforms, making the function execute more number of operations in a given time and time taken by the function to execute should be reduced in the cloud platform. This would help the database community to gain the advantage of serverless functions in handling the data efficiently and can make the database systems fast and efficient.
6. Conclusion and Future Work
7. Appendix

7.0.1 Implementation code

In this section, some code of our implementation is included for reference.

7.0.2 Python flask file for native execution

```python
class INSERT(Resource):
    args = {
        'table': fields.Str(required=True),
        # validate=validate.OneOf(['baz', 'qux'])),
        'key': fields.Str(required=True),
        # 'field' : fields.Str(required=True)
        'field0': fields.Str(required=True),
        'field1': fields.Str(required=True),
        'field2': fields.Str(required=True),
        'field3': fields.Str(required=True),
        'field4': fields.Str(required=True),
        'field5': fields.Str(required=True),
        'field6': fields.Str(required=True),
        'field7': fields.Str(required=True),
        'field8': fields.Str(required=True),
        'field9': fields.Str(required=True)
    }

    @use_kwargs(args)
```

@use_kwarg(args)
# def get(self, table, key,*args):
# return {'Message':table, 'Message2':key,'Message3':field}
def post(self, table, key, field0, field1, field2, field3, field4, field5, field6, field7, field8, field9):
    # If field0 is not None, then read key and field =from redis...
    redis_fields = {} 
    if field0 is not None:
        redis_fields["field0"] = field0
    if field1 is not None:
        redis_fields["field1"] = field1
    if field2 is not None:
        redis_fields["field2"] = field2
    if field3 is not None:
        redis_fields["field3"] = field3
    if field4 is not None:
        redis_fields["field4"] = field4
    if field5 is not None:
        redis_fields["field5"] = field5
    if field6 is not None:
        redis_fields["field6"] = field6
    if field7 is not None:
        redis_fields["field7"] = field7
    if field8 is not None:
        redis_fields["field8"] = field8
    if field9 is not None:
        redis_fields["field9"] = field9
    redis_db.hmset(key, redis_fields)
    print("We're here...")
    return {'Message': table, 'Message2': key, 'Message3': redis_fields}

class READ(Resource):
    args = {
        'table': fields.Str(
            required=True,
            # validate=validate.OneOf(['baz', 'qux'])),
        ),
        'key': fields.Str(required=True),
        }
        'field0': fields.Str(required=False),
        'field1': fields.Str(required=False),
        'field2': fields.Str(required=False),
        'field3': fields.Str(required=False),
        'field4': fields.Str(required=False),
        'field5': fields.Str(required=False),
        'field6': fields.Str(required=False),
        'field7': fields.Str(required=False),
        'field8': fields.Str(required=False),
        }
@use_kwargs(args)
def get(self, table, key, field0, field1, field2, field3, field4, field5, field6, field7, field8, field9):
    # If field0 is not None, then read key and field =from redis...
    redis_fields = {}
    if field0 is not None:
        redis_fields['field0'] = field0
    if field1 is not None:
        redis_fields['field1'] = field1
    if field2 is not None:
        redis_fields['field2'] = field2
    if field3 is not None:
        redis_fields['field3'] = field3
    if field4 is not None:
        redis_fields['field4'] = field4
    if field5 is not None:
        redis_fields['field5'] = field5
    if field6 is not None:
        redis_fields['field6'] = field6
    if field7 is not None:
        redis_fields['field7'] = field7
    if field8 is not None:
        redis_fields['field8'] = field8
    if field9 is not None:
        redis_fields['field9'] = field9
    return {'Message': redis_db.hgetall(key)}
    # return {'Message1': table,'Message2': key,'Message3': redis_fields}

7.0.3 YCSB file for Read, Insert, Update and Scan

In the string url section, first the endpoint is specified followed by the type of operation to be performed.


@Override
def read(table, key, fields, result):
    try {
        String url = "https://europe-west1-serverless-functions-217415.cloudfunctions.net/function-2/read?table=user&key=" + key;
//        if(fields!=null){
```java
for (String field : fields) {
    url += "fields=\"+field+\"&";
}

url = url.substring(0, url.length()-1);
URL obj;
HttpURLConnection con = null;
obj = new URL(url);
con = (HttpURLConnection) obj.openConnection();
con.setRequestMethod("GET");
con.setRequestProperty("Accept", "application/json");
BufferedReader in = new BufferedReader(new InputStreamReader(con.getInputStream()));
InputStream response = con.getErrorStream();
ObjectMapper mapper = new ObjectMapper();
String inputLine;
while ((inputLine = in.readLine()) != null) {
    Map<String, Map<String, String>> object = mapper.readValue(inputLine, new TypeReference<Map<String, Map<String, String>>>() {});
    System.out.println(object.get(key));
    Map<String, String> object2 = new HashMap<String, String>();
    object2 = object.get("Message");
    for (Map.Entry<String, String> entry : object2.entrySet()) {
        object2.put(entry.getKey(), entry.getValue());
    }
    StringByteIterator.putAllAsByteIterators(result, object2);
}
in.close();
} catch (Exception e) {
    e.printStackTrace();
    return Status.ERROR;
} return Status.OK;

@Override
public Status insert(String table, String key, Map<String, ByteIterator> values) {
    try {
        Map<String, String> map = StringByteIterator.getStringMap(values);
        String url = "https://europe-west1-serverless-functions-217415.cloudfunctions.net/function-2/insert";
        String payload = "\"table\":\"User\", \"key\":\"+key+\"";
        for (Map.Entry<String, String> field : map.entrySet()) {
            payload += \":\"+field.getKey()+\":\"+URLEncoder.encode(field.getValue(), "UTF-8")+\"\n        }
        payload = payload.substring(0, payload.length()-2);
        payload += ");";
        System.out.println(payload);
    } catch (Exception e) {
        e.printStackTrace();
        return Status.ERROR;
    } return Status.OK;
```
System.out.println(url);
URL obj;
HttpURLConnection con = null;
obj = new URL(url);
con = (HttpURLConnection) obj.openConnection();
con.setDoOutput(true);
con.setRequestProperty("Content-Type", "application/json");
con.setRequestProperty("Accept", "application/json");
con.setRequestMethod("PUT");
con.connect();
byte[] outputBytes = payload.getBytes("UTF-8");
OutputStream os = con.getOutputStream();
os.write(outputBytes);
os.close();
con.getResponseCode();
jedis.zadd(INDEX, hash(key), key);
}
} catch (Exception e) {
e.printStackTrace();
return Status.ERROR;
return Status.OK;
@Override
public Status update(String table, String key, Map<String, ByteIterator> values) {
try {
Map<String, String> map = StringByteIterator.getStringMap(values);
String url = "https://europe-west1-serverless-functions-217415.cloudfunctions.net/function-2/update?table=user&key="+key+";
for (Map.Entry<String, String> field : map.entrySet()){
    url+=field.getKey()+"="+URLEncoder.encode(field.getValue(), "UTF-8")+"&";
}
url = url.substring(0, url.length()-1);
URL obj;
HttpURLConnection con = null;
obj = new URL(url);
con = (HttpURLConnection) obj.openConnection();
con.setDoOutput(true);
con.setRequestProperty("Content-Type", "application/json");
con.setRequestProperty("Accept", "application/json");
con.setRequestMethod("PUT");
con.connect();
con.getResponseCode();
}
} catch (Exception e) {
e.printStackTrace();
return Status.ERROR;
}
```java
    return Status.OK;
}

@Override
public Status scan(String table, String startkey, int recordcount,
        Set<String> fields, Vector<HashMap<String, ByteIterator>> result) {
    try {
        String url = "https://europe-west1-serverless-functions-217415.cloudfunctions.net/function-2/scan?table=user&key=\"+startkey;
        url+=\"&recordCount=\"+recordcount;
        if (fields!=null){
            for (String field: fields){
                url+=\"fields=\"+field+\"&\";
            }
        }
        url = url.substring(0, url.length()-1);
        URL obj = new URL(url);
        HttpURLConnection con = (HttpURLConnection) obj.openConnection();
        con.setRequestMethod("GET");
        con.setRequestProperty("Accept", "application/json");
        BufferedReader in = new BufferedReader(new InputStreamReader(con.getInputStream()));
        String inputLine;
        ObjectMapper mapper=new ObjectMapper();
        while ((inputLine = in.readLine()) != null) {
            Map<String, String> object2=new HashMap<String, String>();
            object2=mapper.readValue(inputLine, new TypeReference<Map<String, String>>(){}, new TypeReference<
            Map<String, String>>(){});
            for (Map.Entry<String, String> entry: object2.entrySet()){object2.put(entry.getKey(), entry.getValue());
            }
            result.addElement((HashMap<String, ByteIterator>) StringByteIterator.getByteIteratorMap(object2));
            in.close();
        }
    }catch (Exception e) {
        e.printStackTrace();
        return Status.ERROR;
    }
    return Status.OK;
}
```

### 7.0.4 Serverless function

In the host, the IP address of the redis-server machine is given. For the emulator execution, the 'localhost' or '127.0.0.1' is given as the host address. In case of cloud platform, always elastic or external IP address is given.
"use strict";

var http = require ('http');
var redisStore = require('connect-redis');
var redis = require ('redis');
var client = redis.createClient({host : '35.240.65.22', port : 6379});

client.on('connect', function(){
    console.log ('Redis Client connected from function handleGET!!');
});

client.on('error', function(err){
    console.log('Error when connecting from handleGET.. ' + err);
});

function handleGET (req, res){
    let user;
    let key;
    user= req.body.user;
    key= req.body.key;

    client.hgetall(key, function (error, results){
        res.status(200).send(results);
    });
}

function handlePOST (req, res) {
    let key;
    let user;
    //var fields = new Array();
    let field0;
    let field1;
    let field2;
    let field3;
    let field4;
    let field5;
```javascript
let field6;
let field7;
let field8;
let field9;
user = req.body.user;
key = req.body.key;
field0 = req.body.field0;
field1 = req.body.field1;
field2 = req.body.field2;
field3 = req.body.field3;
field4 = req.body.field4;
field5 = req.body.field5;
field6 = req.body.field6;
field7 = req.body.field7;
field8 = req.body.field8;
field9 = req.body.field9;
client.hmset(key, ["field0", field0, "field1", field1, "field2", field2, "field3", field3, "field4", field4, "field5", field5, "field6", field6, "field7", field7, "field8", field8, "field9", field9], function (err, results) {
    res.status(200);
});

exports.hello = (req, res) => {
    switch (req.method) {
    case 'GET':
        handleGET(req, res);
        break;
    case 'POST':
        handlePOST(req, res);
        res.status(200).send();
        break;
    default:
        res.status(500).send({ error: 'Something blew up!' });
        break;
    }
};
```
Bibliography


Bibliography


