Big Data Cloud Computing Infrastructure Framework

A framework for developing reproducible cloud computing infrastructures suitable for big data processing jobs

Project Report

Group dt905e18
Charles Robert McCall
Cristian Viorel Buda

Aalborg University
Computer Science

AALBORG UNIVERSITY
STUDENT REPORT

Electronics and IT
Aalborg University
http://www.aau.dk

Title:
Big data cloud computing infrastructure framework

Theme:
Scientific Theme

Project Period:
Spring Semester 2018

Project Group:
dt905e18

Participant(s):
Charles Robert McCall
Cristian Viorel Buda

Supervisor(s):
Christian Thomsen

Copies: 1

Page Numbers: 98

Date of Completion:
September 14, 2019

Abstract:

Building an environment suitable to handle big data workloads involves using multiple software together to form the end result. We define a software framework as containing a suite of software packages to be used together to form a reproducible environment to run big data tasks on. Each choice of software is justified and its corresponding code is explained, as well as the resulting environment is demonstrated by running experimental big data tasks. The infrastructure is built by leveraging the Google Cloud Platform cloud computing provider to build the hardware. Terraform, an infrastructure manager, is used to communicate with the Google Cloud Platform API in order to programatically build the hardware infrastructure, while the Nix package manager is used to download, setup and configure the software packages. This framework can be used to build similar environments or adapt and further expand the code presented in this paper.

Contents

Preface

Aalborg University, September 14, 2019

Charles Robert McCall
cmccal16@student.aau.dk

Cristian Viorel Buda
cbuda17@student.aau.dk

Chapter 1

Introduction

Big data is is a field that analyzes and processes large amounts of data that cannot be handled by traditional software due to its size. Data is becoming increasingly important for businesses to make good decisions in order to gain a competitive edge in their respective market they operate on. Modern data sets, however, are becoming larger and more detailed, causing new challenges to arise such as storing and making sense of it. Big data can be very large, to the degree of several exabyte, and come in various formats and types of information stored. This data needs to be ingested and stored into a system, analyzed, processed, queried, properly visualized, all relatively fast. The key action being the analysis of the data, with the intention of gaining valuable insights from it, such as trends or being able to predict certain events. Any modern company could potentially benefit from having a big data framework and a culture of analyzing their past records [2].

Setting up the technological environment to handle big data tasks can be difficult. Many tools exist on the market today to make the building of the required technological infrastructure easier, but developers are daunted with the task of choosing the most appropriate software tools. The concept of a software framework is central to large-scale software systems. Frameworks promise developers to be more productive, have shorter development times and higher quality applications. A software framework is defined as an abstraction of some software that can be easily changed to obtain different functionality. Such a software framework provides a streamlined, reusable and standard method of developing software applications. A software framework can contain any type of software to be brought together in order to create a larger application [13].

1.1 Motivation

Big data processing requires computing power greater than that of a single computer's limited hardware resources. Thus arises the need for a more complex solution to solving this problem with a distributed computing approach. Thus, we made use of cloud computing, which allowed us to have a more ambitious hardware infrastructure, with the option for scaling up or out with relative ease. We designed a software framework suitable for building a reproducible environment fit to handle big data tasks.

1.2 Problem statement

Such an environment is built by using multiple software packages to be used together. There exist many such packages, and such arises the problem of which to choose, some may be incompatible with each other. We set out to solve this problem by designing a software framework containing multiple software components working together to form the overall environment required by big data tasks. Moreover, our goal was to make our proposed environment reproducible to another machine easy. The framework should be extensible, with the ability to add new software tools to be installed on the resulting environment. This environment and the chosen packages then need to be validated, with standardized benchmarks proving reliable performance results.

We can formulate three problem statements we want to address with this report:

- Can we create a software framework to form a reproducible environment to run big data performance tests on?
- Can we extend the framework to include alternative software tools to enhance our performance tests by comparing results across software tools
- Can we run a standardized benchmark for reliable performance tests of software

1.3 Overview

This project aims to answer these questions, by presenting a suite of different software that can be used together to form an environment for big data processing to handle large workloads that leverages cloud computing services. This paper goes into explaining the challenges more in-depth, such as the concept of cloud computing and distributed systems and the appropriate software used to develop such a framework and run the tools necessary to perform performance tests.

Chapter 2 introduces papers covering relevant Computer Science literature done by other groups which influenced our decisions. Chapter 3 lays down the knowledge base necessary to understand the project. This includes the concepts and essential information behind the software and services used, as well as theory behind running benchmarks on big data processing software. Chapter 4 presents the software and services used and their corresponding implementation details and development methodologies. Chapter 5 explains the TPC-DS specification and the decisions made on what parts of it to use. Chapter 6 covers our implementation and execution of the benchmark. Chapter 7 Briefly discusses the results of the limited benchmark test. Finally, Chapter 8 is where our experience is discussed and reflected upon, and concluded, and the possibilities of extending the project into the future.

Chapter 2

Related Work

This chapter introduces the papers which were influential in choosing and developing our project.

The paper titled "SQL-on-Hadoop: full circle back to shared-nothing database architectures" by Avrilia Floratou et. al (2014) [6] presents different SQL query processing software and compares their performance. This was done through standardized benchmarks found in the TPC-H toolkit. They go in detail on how they used two data processing tools, namely Hive and Impala. They also focus on the two different file formats, namely ORC and Parquet. This paper gave some insight into what our project may be, but the paper falls short on several factors. They say they are TPC-H inspired, but they do not state how they deviate from a valid TPC-H run, or how they implemented the software and scripts to perform the tests. They do state however that they used a scale factor of 1000 (the smallest valid scale factor), and they were unable to use bigger scale factors due to software limitations of their tools which turns out to be quite relevant for us given we have some resource troubles of our own related to our cloud environment, covered in chapter 4.

The paper titled "On Big Data Benchmarking" [8] talks about the challenges of big data and how to design benchmarks for it. The methodology presented in the paper helped us better understand how benchmarking big data better in order to design our proposed software framework, by choosing the appropriate tools, designing test and data with suitable attributes and testing criteria.

"Why you should run TPC-DS: A Workload Analysis" (Poess, Nambiar and Walrath, 2007) [11] presents the benefits of using the Transaction Processing Performance Council's (TPC) standardized support benchmark specialized in support decision systems. The TPC-DS benchmark is widely used by companies to demonstrate the capabilities of their decision support systems. It goes in detail to explain the three main models, namely related to the data, workload and execution model. It helped us understand the characteristics of the benchmark and its different phases and their impact. The paper also argues that it will replace TPC-H, which was the benchmark used in [6]. Thus, this paper helped us choose TPC-DS as our toolkit to generate data and queries.

Dolstra el. al's 2008 paper titled "NixOS: A purely functional Linux distribution" introduced us to the concept of purely functional operating system and package manager. It helped us achieve the reproducibility within our project we needed by using NixOS as the operating system running on all of our cluster's nodes, as well the Nix purely functional package manager. This paper goes into detail explaining the benefits of such a setup, which include easily configurable and reproducible systems.

Chapter 3

Background Knowledge

This chapter explains the suite of software and services used in developing the project's infrastructure and the comparison to some of their respective alternatives. Section 3.1 introduces the concept of distributed computing and its benefits in the context of big data processing. Section 3.2 covers all the topics related to the project's infrastructure. It presents the concepts of cloud computing and managing infrastructure programatically through an approach called *infrastructure as code*. Two important components in our project are Terraform and NixOS. Terraform, our chosen infrastructure manager is covered in Section 3.3. NixOS and the Nix package manager are explained in Section 3.4.

In order to make meaningful tests and be able to conclude the project with trusted information, Section 3.5 researches and explores existing benchmarking practices/performance evaluation tools, and explores tools to carry out the experiments. This chapter uses information from the "On Big Data Benchmarking" [18] paper, in regards to how benchmarks should be done and how benchmark generators should be created and configured for use in general. Section 3.6 details the process of choosing the right tool for running benchmarks.

3.1 Distributed Computing

Data is being generated worldwide at an increasingly higher rate. This accumulated data eventually needs to be analyzed in order to learn something valuable from it, and do so quickly enough to be a useful pursuit. The benefits of accumulating and analyzing big data is the valuable insights obtained through converting seemingly unhelpful data into knowledge, thus helping with the intelligent decision-making process. Big data is defined by its high volume, velocity and variety that requires specialized software and analytical approaches to successfully convert it into valuable knowledge [9].

A modern solution to this problem is using distributed computing. Distributed computing entails having a network of computers connected to each other in order to pool their resources into solving a common task. It does come with additional setup overhead to distribute the tasks to the network, but the overall benefits make it a viable option. A single computer is limited to its hardware's capabilities, be it memory, processor or disk. Modern computers offer power at a low cost, comparatively to even a decade ago, thus making distributed computing a reasonable solution. In this case, commodity hardware-built computers can be added to the network as another resource, adding to the power of the overall cluster of computers, resulting in power that far exceeds any single computer can generate. This is relatively cheaper than just replacing one computer's hardware components with better ones, also known as scaling up [10].

Distributing and executing data analysis tasks over a network of computers requires a lot of computational resources. Even with the best network connection between the nodes, the transfer rate is still inferior to that of RAM. However, it's performance can be comparable with that of disks, making distributing the data storage across many computers viable for big data tasks. Once the large data set has been distributed across the network, it is viable to distribute the processing of tasks as well. Another benefit to having a cluster of computers is that it prevents having a single point of failure. If one of the computers were to fail, the rest of the network can still continue working. This usually requires one node's data to be replicated on more than one node, in order to save a copy, if one were to be faulty [10].

3.2 Infrastructure

The infrastructure is the foundation to any large scale software-based project, but especially in big data projects where lots of computation and storage resources are required. As specified by Jacobs [10], the complexity can increase tenfold when talking about a distributed computing environment. Requirement gathering was the foundation of making a decision on our project's infrastructure. We needed to answer the question of what software and hardware our experiments will run on, thus choosing the suitable software tools in order to build an infrastructure fit for running all our required big data tasks efficiently, and even taking the possibility of scalability and expanding the project's resource requirements for potential future work. Due to lack of hardware and resources, we decided to opt for a cloud computing infrastructure.

3.2.1 Cloud computing

Modern companies, especially technology based ones, need flexible, scalable and cost effective computer systems. Cloud computing fulfill these needs through the delivery of on-demand computing services to fit the needs of different kinds of companies, especially small to medium sized ones that do not have the capital to sustain the traditional hardware infrastructure and specialized operations development team of employees, commonly found in large technology companies. This broadened accessibility is due to the fact that it adopts a "pay-as-you-go" revenue model, allowing for smaller companies to compete in their respective markets without the need to commit to the upfront costs of hardware infrastructure acquisition. Cloud computing resources are remotely available computing resources where the user can make use of it with minimal or no hardware expertise. The *cloud* refers to the total hardware and software resources made available by a vendor's data center. Most vendors rent out their cloud computing resources to customers in small increments for a time, allowing the user to pay only for the resources they used, usually by the hour [19].

There are three major benefits to cloud computing:

- **Available hardware variety**: users being able to easily rent out a lot of different hardware with various power and price [1].
- **Renting system on-demand**: users can use and change the hardware as needed, without needing to commit to big hardware purchases up-front or even need to have the technical expertise to setup and maintain the hardware infrastructure. The benefit is ease of scalability of the users' hardware infrastructure [1].
- **Pay-as-you-go payment model**: users pay only for what hardware they use and how much they use it. The benefit is lower overall costs for both client and vendor providing the cloud computing solutions. When a client decides they want to stop using a certain cloud computing resource, the hardware gets freed to be rented out to another client [1].

3.2.2 Infrastructure as code

Tech companies used to have different types of technical specialized teams, one type handling *development* of applications and the other handling the *operations* of the hardware required for the software to run on. A typical development team would write code, build a working software application, then pass it to the operations team for them to handle the deployment side of the process. Much of the operations' team work was done manually, physically buying and setting up hardware, then installing the software the development team created to ensure it works properly. This usually involved installing its software dependencies as well. The more the company grows, the more complex and human error-prone this organizational structure is [3].

An *infrastructure as code* (IAC) approach entails defining, deploying and updating infrastructure through code. This can be anything such as servers, databases, configurations, automated testing, deployment processes etc. This programatic approach allows users to consider aspects of operations as pieces of software to be developed, including the ones representing hardware [3].

IAC tools can be split into four categories:

- **Ad hoc scripts**: code written with the purpose of automating some tasks requiring multiple steps (e.g. installing dependencies and starting up some service or server). The script can be then executed on a server. Since writing such scripts in programming languages, such as Bash or Python, can be easy if the tasks are simple, when it comes to more complex ones, special IAC tools should be used that make it easier to write such complex scripts.

- **Configuration management tools**: distribute, install and manage software on existing servers. Such specialized tools, such as Ansible, enforce a consistent structure that help developers stick to code conventions in order to improve readability.

- **Server templating tools**: An alternative to configuration management where an *image* of a server is created. This image contains everything in a server: operating system, installed software packages, existing files etc. A popular example of such a tool is Docker. There are two types of tools that work with images: **Virtual Machines**, where an entire computer's hardware is emulated by a host computer. Any software running on the virtual machine is isolated from the host machine. The second type is **containers**, where a user space is emulated on the same machine, by using a container engine such as Docker. Software running in a container is isolated to this user space, but shares the same hardware as other user spaces, making startup of a container way faster than that of a virtual machine.

- **Server provisioning tools**: actually create the servers, databases, queues, load balancers and the rest of the infrastructure components. Example of such a tool is Terraform [3]. Such tools work well with a cloud provider's API.

Using an infrastructure as code approach improves the software delivery process and offers multiple benefits:

- Self-service: developers can handle their own deployments. Relying on one system administrator is no longer needed.

- Speed and safety: Since the deployment process is automated, it will be faster, safer and easier to repeat as many times as needed than executing the whole process manually.

- Documentation: the state of the infrastructure can be represented in source files that anyone can read, without needing specialized system administration knowledge.
- Version control: committing the IAC source files to version control system such as GitHub can make it easy to keep track of the history of changes to the infrastructure, which can be a great benefit when debugging problems caused by the added changes.
- Validation: since the infrastructure state is all code, code reviews sessions and automate tests can be implemented to prevent bugs.
- Reusability: tested infrastructure code can be packaged and reused [3].

Configuration management versus provisioning

Tools such as Ansible handle configuration management, while Terraform is a provisioning tool. They are not entirely different from each other, as a configuration management tool can do some provisioning and vice versa. Both Ansible and Terraform can be used to deploy servers and run configuration scripts on them. An alternative is to create an image with Docker, a server templating tool, then provision the required infrastructure that the image needs to run on by using Terraform. In our case, we use Terraform for provisioning and Nix to configure each machine and setup software packages [3].

Immutable infrastructure

There are two different paradigms when it comes to infrastructure: immutable and mutable. Configuration management tools such as Ansible [3] adopt the mutable infrastructure paradigm. When a change is made to the infrastructure, such a tool will schedule an update on the servers in order to apply said change. Updating servers frequently can lead to configuration bugs and inconsistencies, leading to differences between servers. In such a case, worker nodes would no longer be clones of each other [3].

Terraform solves this problem by adopting the immutable infrastructure paradigm. If machines are deployed as Docker images via Terraform, changes are not updates, but completely new instances of servers. Each change means a new immutable image file deployed on a new server. The benefit to this approach is a decrease in bugs and inconsistencies, and overall easier deployment of different versions of software. Since the changes are not updates, there is no update history. This helps with automated testing, as a test environment will be no different than a production environment with a long history of updates that is hard to replicate. The downside of this paradigm is that rebuilding an image then redeploying servers with this new image can take a long time, although it is done automatically, without the need for developer input after the process has started [3].

Declarative programming paradigm

A programming language adopting the procedural programming paradigm, such as Ansible's, requires specifying step by step instructions until an end state is reached. Terraform adopts the declarative programming paradigm. The the developer specifies the end state through code, then leaving Terraform to create the optimized plan containing the step by step instructions. The benefit is the reduced complexity in developing infrastructure and increased productivity [3].

The difference between the initial code does not differ that much. The following code snippet declares 10 machines to be created in the langauges pertaining to each IAC tool.

```
Terraform:
resource "google_compute_instance" "worker" {
  count        = 10
  name         = "${var.env}-gcespark-worker-${count.index}"
  machine_type = "n1-standard-2"
  zone         = "europe-west1-b"
  }

Ansible:
-gcp_compute_disk:
  count: 10
  name: "${var.env}-gcespark-worker-${count.index}"
  machine_type:"n1-standard-2"
  zone:"europe-west1-b"
```

They are very similar and will do the same thing, as it is the initial task to be executed. Things differ greatly when attempting to make changes. To give a simple example, if the infrastructure needs change and additional worker nodes need to be added, the way this change needs to be implemented differs between the two IAC tools. More complex cases would become even harder to develop and maintain with Ansible, as the developer needs to be aware of the already existing infrastructure components, their past history, version, availability, dependencies etc. Terraform is a much more viable alternative in keeping the code complexity to a minimum, as we did in our project.

In Ansible, if the *count* variable were to be changed to take the value 15 then the code executed once more, it would deploy 15 new servers on top of the already existing ones. This adds to the complexity of development, as the user must keep track of already existing infrastructure before running code. Thus, the variable should be changed to 5 in order to have 15 nodes [3].

In Terraform, the changes can be done on the original code, changing the variable to 15 then executing *terraform apply*. Terraform then evaluates the specified end state then figures out how to get there. This is possible because Terraform is aware of previous states it created, forming a plan to execute tasks in order to reach the desired end state based on the current state. This has the benefit of being much easier to develop and maintain, saving both time and effort to the developer [3].

3.3 Infrastructure as Code with Terraform

The solution to making the technical infrastructure easier to manage is using Terraform while opting for a cloud computing solution. It is an infrastructure manager, which takes an *infrastructure as code* approach, letting the developer manage their technical infrastructure, or in our case the entire cloud cluster, programmatically. This powerful tool allows for both easy infrastructure setup and automating building or destruction of any cloud component. Since an organization's technological infrastructure can quickly become complex over any period of time, thus the need for such a tool arose.

Terraform lets the developers declaratively define configuration files. As common with the declarative programming paradigm, the programmer writes code that specifies *what* they want and the compiler handles *how* to get there. In Terraform's case, the configuration files specify the programmer's desired resulting cloud infrastructure. With such modern tools, the term *DevOps* came to be, which refers to the set of processes, ideas and technologies to improve the operation side of software development. Terraform aims to solve the problem of automation in the DevOps space [3].

Terraform is compatible with many cloud computing providers, such as *Google Cloud Platform* (GCP), which we chose for this project. As GCP offers a variety of hardware and services, Terraform offers the option to manage all of them programmatically, thus allowing a small team of developers, such as ours, to orchestrate our project's cloud infrastructure with relative ease, compared to if we had to do everything manually through GCP's web interface every time we wanted to make a change [3].

3.3.1 How Terraform works

Terraform, which is an open-source tool, is designed for infrastructure deployment. Even a single developer can be productive and use it to build complex infrastructure from a single machine, like a laptop, without requiring additional infrastructure. That is because, behind the scenes, Terraform communicates with different cloud computing providers via their API. In this project's case, Terraform makes API calls to GCP on our behalf, by having access to the project's GCP secret key for authentication [3].

Terraform calls the APIs specified in text files named *Terraform configurations*. The code within these configuration files specify the desired infrastructure, hence the term *infrastructure as code* [3].

The following code snippet shows a basic API call within our project via Terraform code.

```
resource "google_compute_instance" "master" {
  name         = "${var.env}-gcespark-master"
  machine_type = "n1-standard-2"
  zone         = "europe-west1-b"

  boot_disk {
    initialize_params {
      image = google_compute_image.nixos_1809.self_link
      size  = 40
    }}}
```

Terraform code is user readable. In this snippet above, Google Compute Instance is defined with the name *master*. It is a Terraform *resource*, which acts as a singular component. It communicates with GCP's API, more specifically declaring that the desired machine should have the specified name (with the resource name interpolated as a substring), machine type (GCP offers many different types) and its zone. A separate boot disk is declared with the image, which is the NixOS Linux operating system, and the size in gigabytes of said disk. This code declares the machine that is the master node, which is unique [3].

```
    resource "google_compute_instance" "worker" {
  count        = 10
  name         = "${var.env}-gcespark-worker-${count.index}"
  machine_type = "n1-standard-2"
  zone         = "europe-west1-b"

  boot_disk {
    initialize_params {
      image = google_compute_image.nixos_1809.self_link
      size  = 30
    }}}
```

This code snippet above handles the creation of the worker nodes. In this case, an additional variable, *count*, can be used to specify how many worker nodes should Terraform build. Since it is only configuration files that Terraform needs, these instance configurations are split into two different files: master.tf and worker.tf, where *.tf* is as Terraform's file extension. When attempting to create the build plan, Terraform will parse the configuration files. Upon parsing the worker configuration file, it will create the declared number of clones of machines configured according to this file. Thus, scaling is made easy, reducing adding new identically configured worker nodes to the cluster as trivial as changing one line of code before executing the rebuilding process. Some changes require drastic code refactoring, such as switching to a different cloud computing provider. Not all providers offer the same resources or services. In this case, Terraform's documentation must be read in order to refactor the configuration files to be compatible with Terraform's API syntax corresponding to the new provider. Terraform will analyze what current the infrastructure is and compare it with the newly added changes, form a plan of tasks it needs to execute then build or destroy resources until the newly added changes are implemented. Configuration files contain many different resources, with dependencies between them, as well as order they must be built, all of which Terraform manages behind the scenes. This lowers complexity and enables increased productivity when developing software products. To deploy the programmatically specified infrastructure, the **terraform apply** command is executed. This prompts Terraform to parse the code then form a plan containing executable API calls to the specified cloud computing provider [3].

Team collaboration is made easier by committing Terraform source files to version control. When a team member wants to change some aspect of the infrastructure, they must only modify the code within the infrastructure's source files, rather than needing to manually update through the cloud provider's web interface. These changes can be treated like any other newly developed feature, validating them through code reviews, automated tests then committing the updated code to the appropriate version control, then finally executing *terraform apply* to have Terraform deploy the changes [3].

3.3.2 IAC tools: Declarative vs procedural

The previous example of increasing complexity due to procedural paradigm approach of an IAC tool such as Ansible, highlights the major drawbacks of procedural code when working with infrastructure as code:

- Procedural code does not specify the entire state of the infrastructure. The user must know the correct order in which Ansible templates are applied, as a different order would cause errors or a different infrastructure to be built. This is not apparent from reading procedural code alone. Handling such a codebase requires knowledge of past change history.

- Limited reusability. It is harder to reuse procedural code, as knowledge about the current state of the infrastructure is required, which is changing constantly. This causes previously used code to become unusable on a newer version of the infrastructure. This adds lots of development overhead, as the codebase grows, it needs to be refactored and tested after every infrastructure change [3].

Terraform solves these problems by adopting the declarative programming paradigm. Thus, the code always shows the state the infrastructure is in at that time and knowledge of its history is not required. By simply observing the code, it is relatively easy to understand the infrastructure's deployed components and their respective configuration, being able to disregard its past changes. Reusable code is easier to write, as the current state of the infrastructure does not need to be taken into account. When developing with Terraform, the user must only write declarative code that describes the *desired state of the infrastructure*, thus reducing knowledge overhead and complexity. Terraform codebases remain relatively small and simple to understand even after lots of changes and adding new infrastructure components. The declarative programming paradigm, however, lacks the procedural paradigm's expressive power. An example of this is the lack of basic logic structures such as if-statements or loops, which could pose problems when writing reusable code. However, Terraform does provide some useful primitives such as input and output variables, modules, *count* and interpolation functions. These components aid the developer in writing easily configurable and modular code while still using a declarative programming paradigm.

3.4 NixOS: A Configuration Based OS

Once the hardware part of the infrastructure is set up, the next question is what software should the machines run? This includes the operating system, system configurations and software packages installed.

System configuration tools usually adopt an imperative model. This approach implies *stateful* system administration actions. When upgrading software packages or making changes to the system's configuration files, the state of the system changes, destroying the previous state. This can cause several problems, explained in the following subsection [5].

This section presents a solution to these problems, namely adopting a *purely functional system configuration* model. This model handles the building of static parts of a system such as its configuration files and software packages through immutable *pure functions*. This model was implemented by Dolstra and Löh [5] as a purely functional Linux distribution called NixOS. Its package manager, Nix, is used to build system configurations from code written in a functional programming paradigm. In our project, we made use of both NixOS and Nix in building our framework and setup our infrastructure to be reproducible.

The purely functional model has several benefits:

- **Immutability**: The static parts of a system are immutable. After they get built, they never change, but the system configuration does. The system then gets rebuilt based on the new configuration. Thus, the system is built deterministically.
- **Reproducibility**: Due to the deterministic build process, such a system can be reproduced on another computer.
- **Rollback** to a previous configuration is easy, as past configurations are not overwritten and are still available.
- **Statelessness**: configuration actions are predictable and easier to manage, as they won't fail or introduce bugs due to the state the system was in, as it is possible in stateful systems [5].

3.4.1 Drawbacks of imperative package management

Imperative Unix package management tools such as Debian's *apt* adopt an imperative model. This model implies that such tools change the state of the system, thus having stateful actions. The *apt* package manager stores each package's files in file system hierarchies typical of Unix-based systems (e.g. directory such as /bin). When these packages would be updated, the previous version would be overwritten with the new one. Such an overwriting update of one package could cause dependency problems. If another package depends on the previous version, updating to a new version will break the second package. Thus, in an imperative Unix system, the file system and registry are used similarly to how a programming language would use *mutable global variables* [5].

The stateful nature of such systems make it difficult to have different versions of a package installed at the same time. Having two packages that depend on different version of some other third package would require manually installing separate versions of said package in separate locations with different names. This creates a lot of management overhead for the system user [5].

Building packages are also stateful actions. When building a binary package using an imperative package manager, the process is described by a specification file. This file contains the build actions to be executed, in a specific order, in order to successfully construct the package. The specification file must also contain information regarding the package's dependencies, which cause two problems to arise:

- If the specified dependencies are **incomplete** and does not mention one dependency, a machine with that package will be able to build the package binary. However, another machine without this "hidden" dependency installed won't be able to build the package, and will not return any descriptive error, thus failing the process unexpectedly without further information to the user.
- **Nominal** dependency specifications make reproducibility harder to achieve. Thus, dependencies are specified just by their names. The package manager can successfully build the package binary if the required *minimum* version of some package it depends on is installed, however, it has no knowledge of the exact version it used. For example, if the dependency specification document mentions a version of Python 2.7 and above is required to build some package binary and the package manager succeeds, it has no knowledge of the exact Python version used. Therefore, the building package building process cannot be deterministically reproduced [5].

General system configuration files fall outside the scope of imperative package managers such as Red Hat's RPM. When upgrading a package, these configuration files cannot be overwritten according to the new version, as the user could have change it themselves manually. The package manager is not aware of this change. Ignoring these files upon package changing or merging the user's changes with the package manager's changes aren't the most optimal solutions. Some packages contain post-install scripts, with stateful actions, that attempt to handle this problem. The result of executing many of these ad hoc is a system configuration that is hard to rollback to a previous one or reproduce on another machine. Nix, a purely functional package manager, is a viable solution to these drawbacks [5].

3.4.2 Nix: a purely functional package manager

NixOS's uses a purely functional package manager named Nix, researched and developed by Dolstra et. al in 2004 [4], four years prior to the NixOS project [5] in 2008. The concept of purely functional package management implies the use of a functional programming paradigm when building packages. In this case, such a function's output depends purely on its input, the function arguments. Once a package is built this way, it will never change [5].

At Nix's foundation lie *Nix expressions*, a purely functional language, with lazy evaluation and dynamic typing. These Nix expressions are used to describe graphs of build actions, also named *derivations*. A typical derivation contains three components: a build script, a collection of environment variables and a collection of dependencies, which are just other derivations that it depends on. When building a package, the first step of the process is recursively building all of its dependent derivations before proceeding with executing its build script with its corresponding environment variables [5].

The following code snippet represents an example of a Nix expression. The package described is *xmonad*, a Haskell-based tiling X11 window manager. At the physical level, it is a file named "xmonad.nix".

```
1  {stenv, fetchurl, ghc, X11, xmessage}:
2  let version = "0.5"; in
3  stenv.mkDerivation ( rec {
4     name = "xmonad-${version}";
5     src = fetchurl {
6        url="http://....org/.../${name}.tar.gz";
7        sha256="cfcc4501b000fa740ed35a5be87dc012...";
8     };
9     buildInputs = [ghc X11];
10     configurePhase = ''
11        substituteInPlace XMonad/Core.hs --replace\
12        '"xmessage"' '"${xmessage}/bin/xmessage"'
13        ghc --make Setup.lhs ./Setup configure --prefix="${out}"  '';
14      buildPhase = '' ./Setup build '';
15      installPhase = ''  ./Setup copy
16          ./Setup register --get-script '';
17       meta = { description = "A tiling window manager for X"; }
18  })
```

The entire Nix expression acts as a function. The very first line describes the set of arguments it takes as input, while the third one specifies the resulting output, namely a derivation. Name/value pairs are named *attributes*. A set of attributes is surrounded by braces. In Nix expressions, they are the most important part of specifying the derivation details. In the example above, lines 3-18 form the Nix expression's attribute set. The *rec* keyword defines it as recursive, thus allowing an attribute inside the set to refer to another attribute. The first line defines the Nix expression as an anonymous function. The specified packages within the braces are the required attributes when calling this function, also known as its dependencies. These packages will be used in the writing of the Nix expression's attribute set. The function *stenv.mkDerivation* at line 3 acts as a helper function that simplifies the process of specifying build actions by handling boiler plate code which would normally be required when writing build actions. Such boiler plate code include simple actions such as unpacking archives, generating makefiles, compiling packages, installing them in specific directories etc. *mkDerivation* allows different phases of the package building process to be overwritten with shell commands. The *configurePhase* on lines 10-13 configure the package with Bash commands. The double single quote allows for multi-line commands. When calling *mkDerivation*, every attribute but *meta* is passed as an environment variable to the build script.

Line 2 defines the environment variable *version* to take the value "0.5", which is then used by mkDerivation to name the derivation "xmonad-0.5" through interpolation. On line 9, the *buildInputs* environment variable contain the paths to *ghc* and *X11*, which are dependencies to the whole function, as specified in line 1. The *out* variable in line 13 signifies the location in the filesystem where the package will be installed in [5].

Thus, the expression describing how to build *xmonad* is a function that, when called with the same values for its arguments, will return the exact same instance of the *xmonad* package. Changing anything about the expression will result in a different instance. Different calls to this function will not interfere with each other [5].

Another expression can make use of this *xmonad* build function, as demonstrated in the following expression named "all-packages.nix":

```
rec {
    xmonad = import .../xmonad.nix {
        inherit stenv fetchurl ghc X11 xmessage;
    };
    ...
}
```

This code snippet specifies an attribute set containing the attribute named *xmonad* that is bound to the result of calling "xmonad.nix" as a function. The *xmonad* package can be now installed through the terminal by running the following command:

```
$ nix-env -f all-packages.nix -i -A xmonad
```

The *xmonad* derivation will be built, along with all its dependencies then finally putting the package's binary file in the user's filesystem path [5]. To uninstall this package, the following command is to be run:

```
$ nix-env -e xmonad
```

3.4.3 The Nix store: storing packages the purely functional way

The build scripts from Nix expressions install packages wherever the path that the *out* environment variable points to. Nix then stores these packages in a purely functional fashion. This implies that packages must be separated by their different instances as to not interfere with each other [5].

Nix uses a specific file hierarchy to store all of its packages as immutable children of its store. At the physical level, this means storing each package in their specific directory under /nix/store. The packages' directory names contain 160-bit cryptographic hashes generated from the inputs (attributes of the derivation) used to build the corresponding packages. For example, after Nix computes this unique hash for the package previously mentioned, *xmonad*, it will be store the package's binary files in *out*'s new value, /nix/store/8dpf3wcgkv1ixghzjhljj9xbcd9k6z9r-xmonad-0.5/.

Packages are managed by each user, through the concept of *user profiles*. Each user has their own packages managed by the Nix store, separate from other users. This allows for atomic package upgrading and easy change rollbacks. At the physical level, these profiles are stored in /nix/var/nix/profiles [5].

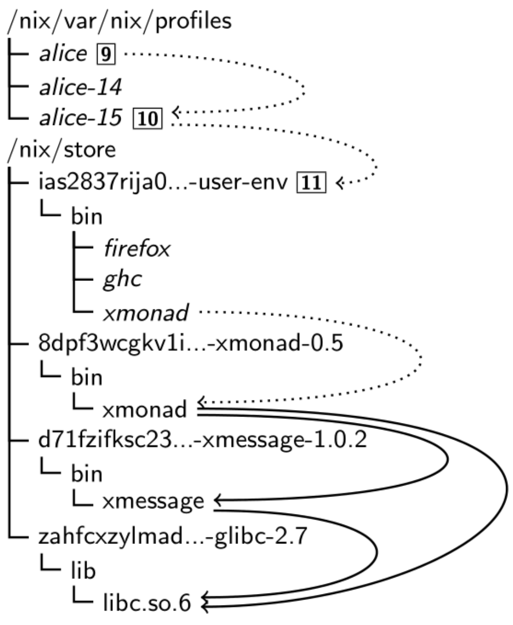
/nix/var/nix/profiles
alice 9
alice-14
alice-15 10
/nix/store
ias2837rija0...-user-env 11
bin
firefox
ghc
xmonad
8dpf3wcgkv1i...-xmonad-0.5
bin
xmonad
d71fzifksc23...-xmessage-1.0.2
bin
xmessage
zahfcxzylmad...-glibc-2.7
lib
libc.so.6

Figure 3.1: Example of a Nix store and Nix profiles [5]

Figure 3.1 illustrates a partial version of an example Nix store. It contains the *xmonad* package and its dependencies. The solid arrows signify the references between files. Slanted names represent symlinks and dotted arrows their targets. In this example, the filesystem path of the user "Alice" would reside in /nix/var/nix/profiles/alice/bin. Upon installing a package, a user environment is built via the package *user-env*. This user environment contains the symlinks to all the corresponding user's installed packages. For example, the symlink /nix/var/nix/profiles/alice-15 (or some other number after the user's name) is created, which simply points to that specific user environment. /nix/var/nix/profiles/alice then is updated in order to point to the latest user environment's symlink (alice-15). Rollbacks simply require the user profile to change the symlink to a previous one (e.g. alice-14) [5].

Such a purely functional package manager has several benefits:

- The Nix store prevents incomplete dependency specifications. Packages are stored outside any default search paths where they can be found by the system attempting to build a package. If the building process of a package has an undeclared dependency, it will always fail.
- References are scanned, thus allowing Nix to detect runtime dependencies. This means all dependencies are known, and it is possible to deploy packages to other machines and have them work.
- The immutability of packages allow for multiple version of the same package to be installed. If one package requires glibc-2.7 and a new package is installed, which requires glibc-2.8, the first package will keep using version 2.7, while version 2.8 is installed for the other package to use. This prevents "dependency hell".
- Since all runtime dependencies are known, the garbage collector can delete any remaining unused packages to be deleted automatically.
- Upgrading packages to newer versions leave the previous intact. This means that rolling back to a previous version, given that it was not intentionally deleted, a simple operation [5].

3.5 Performance Evaluations through Benchmarking

This project needs to compare the performance of Parquet and ORC fairly to conclude some meaningful and trusted insight into which file format to use for some non-arbitrary use cases. This makes using industry standards in performance evaluation essential.

Evaluating the performance of big data systems, gives information about the expected performance of an execution of a specific instance of hardware and software. The performance metrics evaluated, cover a wide range of measurable factors, some of which are: execution time, scalability, resource utilization and energy efficiency.

Frameworks exist to perform these performance evaluations, which are called benchmarking frameworks. The purpose of these frameworks is to create a structured and standardized suite to perform tests fairly on a range of big data tools with various configurations in order to compare performance. Different frameworks specialise in providing different insight into the various aspects of big data. Some of the main factors of big data that are benchmarked include: data processing engine, file system, disk technology, network interconnects and memory management. Specifically, this project is focusing primarily on the file formats as part of the file system factor, namely Parquet and ORC.

3.5.1 Big data benchmarking creation and approach

In order to utilize a benchmarking suite effectively, it is important to understand the process of benchmarking, and what the important criteria is, in order to make effective use of it.

The benchmarking process explained in [8], categorises benchmarking into five steps as seen in Figure 3.2.

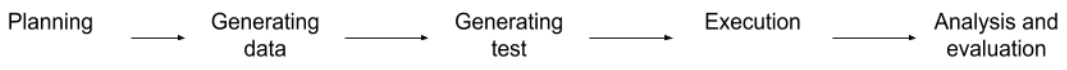

Figure 3.2: The 5 steps in the benchmarking process [8]

3.5.2 Planning

The planning step determines the high levels aspects of what the benchmark tests will do. The objects to be tested and compared are decided, the appropriate domain and use cases are identified, and the evaluation metrics applied to the results are set.

3.5.3 Generating suitable data

Big data is categorised into four dimensions called the **4V's**, which are: **volume, velocity, variety and veracity**.

Volume is the size of the data, such as terabytes, petabytes, ect. Given different workloads of a system, it could mean different things. If the system processes plain/flat data, only the size of the data is important, however if the system has alot of connectivity between its data like a social network, then the volume also includes the number of connections between the data.

Velocity represents the speed of generating more data, updating the data, and processing the data. Generation is measured through a data generation rate, such that you divide the size of the data by the time taken to generate it to determine the hourly rate to produce the data. Real time updating is required by dynamic systems with many users making changes to content, a data updating frequency rate is used to benchmark the update velocity. Systems can also receive constant streams of data that need to be processed for ingestion in real time, velocity is measured as the processing speed for that data.

Variety is the range of data types, and the sources containing them. Data can be structured, semi-structured or unstructed, and the various sources may contain many different data types, nesting and the ability to evolve the schema and update data. Therefore benchmarking data generators are required to support the entire spectrum of sources and data types to enable generation of synthetic data that closely resembles real data, and emulates its behaviour.

Veracity is how closely the data reflects real raw data to be used in a production system. A benchmark is as credible as the veracity of its data, and therefore it is important to understand the data that needs to be created to satisfy a credible benchmark. The paper continues to explain that while some benchmarks use real data, it is often problematic to acquire suitable data to satisfy benchmarking data requirements (veracity) due to the scarcity of suitable public big datasets. The consensus is to generate synthetic data to match real data as closely as possible in order to satisfy the 4V's.

3.5.4 Generating tests

Big data is a diverse and rapidly evolving domain that is composed of many non-trivial parts. Designing and choosing meaningful tests to benchmark a specific application can be challenging given various workload cases for a typical system. The approach is to select the most general use cases, but they must cover the variety of workloads the system can experience. Workloads and their tests are generated from a functional and system view of the systems.

The functional view abstracts the behaviour of a workload, into a general set of instructions which can be applied to the applications being benchmarked to allow for a fair comparison. The abstraction is system independant like pseudo code, such that different systems like a DBMS and a MapReduce system, can understand what is required of them to execute in their own syntax to perform the same instructions respectfully. This is achieved by identifying and abstracting operations of a system, like *select* from a DBMS, and the equivalent "select" operation from MapReduce into a new abstract function that can be applied to both systems to cause the same behaviour. Then abstracting workload patterns which are combinations of those abstracted operations to achieve complex data processing functions, which are basically a ordered set of abstracted operations.

The system view is then applying the abstracted operations or patterns across multiple systems to allow for a comparison on their respective outcomes as they will have performed the same test.

3.5.5 Execution + Analysis and evaluation

Execution involves performing fair benchmarking tests to meet all the varied execution requirements that are experienced across the different tools used in big data. A benchmark generator must be able to adapt to different data sources, such as providing format conversion to transform source data into the required format for the tests. A big data benchmark test may require an entire software stack of tools to compare different software pipelines, therefore benchmark tools must be portable to enable efficient use of all the tools in combination to constitute specific software stacks for testing and comparison. Extensibily of a becnhmark tool is imperative to stay fair and useful as tools develop and new tools and techniques become available requiring support of new use cases. Usability is required for a good user experience for ease of configuration and deployment. Finally, usable benchmark tool that meets these prior requirements promote ease of analysis and evaluation through metric information made available after the tests have been completed.

3.5.6 Summary of the findings on benchmarks

Various benchmarking suites exist, which support data and test generation, execution, evaluation and analysis for big data tests. We have already identified the software tools we want to use, which are Parquet, ORC, Spark, Hive and Impala, within containers on GCP. Going forward, we still need to decide on: the use cases of the system, metrics to be applied for evaluation, how the data will be structured, and their appropriate tests, and the specific configurations for execution.

3.6 Selecting a Benchmarking Tool

Many benchmark suites exist, with the variety of choices typically designed to fill some niche. The niche option catered to this project would be Spark Bench as it is designed for benchmarking specifically on Spark. However, even Databricks when benchmarking Spark, uses TPC benchmarks as opposed to the benchmark project they worked on, Spark Bench; as the TPC benchmarks are industry standard, due to collaboration of many big companies and extensive suites capabilities. Therefore *Transaction Processing Performance Council* (TPC) is an obvious first choice when deciding on a benchmarking suite.

3.6.1 TPC

TPC is a non-profit corporation composed of many big companies like Cisco and Microsoft[17]. They have standardised benchmarking, and created many different and specialised tools throughout the years to measure the different aspects of database and transaction processing systems. Their benchmarks are designed on the most general cases of real world setups to ensure the results fit the majority of real world use cases, and allow for reliable result comparison for real systems.

Three benchmarks from TPC are being evaluated in this report as they are the only three that deal with big data and may be suitable for the project. All three use a retail supplier as their use case for generating data, with a few differences in their schema and the data generated. The benchmarks are designed to transform operational and external data into intelligence, and thus the data is a snowflake schema of normalised tables like an OLTP system. The data generated is given a scale factor value which generates all fact and dimension table data to fit within the corresponding data size which is roughly: 1TB, 3TB, 10TB, 30TB or 100TB[14].

The data generated is designed to be as close to a real production system as possible, and fit for the use cases of the particular benchmark, e.g. TPC-DS generates all structured data, whereas TPCx-BB generates a mix of structured, semi-structured and unstructured data. Data is generated as a flat text file which is either generated directly into the *System Under Test* (SUT) which becomes part of the benchmarking progress, or it can be generated outside the SUT, in which a ETL like process moves the data into the SUT and becomes part of the benchmarking process. The data generated does not contain a header, as it is intended to be loaded into a system prepared with the appropriate schema.

The SUT is the collection of configured components you want to benchmark, e.g. HDFS and Spark. Each TPC benchmark contains a driver which interfaces with the SUT to perform the benchmark. The driver submits queries to the SUT within a session and measures their execution times. The driver may have multiple sessions running on a SUT, and a session is the full benchmark process to be applied to the SUT; which is a load test, followed by two performance tests. The load test is is performed to get the SUT ready to perform the performance test. It consists of data generation and an ETL like process (defined in 5.1.1). The Performance Test consists of two Throughput Tests, each executing the queries as a query stream (which is running all the queries for that benchmark sequentially), and finally producing the results of the session.

TPC-DS

The TPC-DS is for Decision Support. This benchmark is primarily used on big data workloads to turn data into business intelligence[14]. The data generated by this benchmark suite uses a tool within the specification folder called 'dsdgen' (DS - data - generator). The data is structured as it is emulating a relational database, which is to be Transformed and Loaded into the SUT. The generated data contains redundancy to emulate a schema that has evolved over the years, and it also contains NULL values in the majority of its columns. There are seven fact tables, and many dimension tables.

A *decision support system* (DSS) exhibits multiple types of queries for various reasons such as reporting or data mining. TPC-DS has 99 query templates, which fit into one of four categories: Reporting queries, Ad hoc queries, Iterative OLAP queries, and Data mining queries.

TPC-DS uses three primary metrics to benchmark a system: A Performance Metric which evaluates the throughput time of the 99 queries in the query stream, a Price-Performance metric which determines the cost of owning and running the system (all the individual components in the entire SUT such as hardware price, license costs, costs of running hardware/software and maintenance costs), and system availability date in order to ensure the system is commercially available, and not some custom implementation.

TPCx-BB

TPCx-BB is a benchmark created based on the paper "BigBench: Towards an Industry Standard Benchmark for Big Data Analytics"[15]. It uses a schema inspired by TCP-DS (retail snowflake schema), which consists of five fact tables and many dimension tables.

TPCx-BB has 30 queries, ten of which are based on TPC-DS query templates. All the queries are big data analytic use cases, though it differs from TPC-DS in that the data in the various tables can be structured, semi-structured and unstructured data.

Data in TPCX-BB is generated by PDGF based on the scale factor, directly into the SUT and is timed as part of the benchmark process / load test. Like TPC-DS, the majority of the columns can contain null values. TPCx-BB uses the same three metrics as TPC-DS, however it has an optional metric that measures the energy performance of the SUT as throughput divided by energy.

TPCx-HS

TPCx-HS (Hadoop Sort) is a benchmark focused on the execution engines MapReduce and Spark[**tpcxh**]. The benchmark has a data generation tool called HSGen that uses the same scale factor as the other benchmark suites covered, however, there is not a documented schema. This is because the benchmark is intended to sort a dataset in a total order, and does not include any queries.

There is only a performance test, unlike the other two with a load test. Data is first generated with HSGen, and then compliance checked with HSDataCheck (which is not part of the performance test). The performance test is carried out on HSSort, which sorts the generated data into total order based on a another benchmark called TeraSort (TeraSort is a benchmark that sorts randomly distributed data, measuring metrics like time of execution). Finally HSValidate is used to validate the output is correct.

The same metrics are used as explained in TPCx-BB, including the optional energy metric. The query throughput metric in DS and BB is of course adapted to sorting and measures the effective sort throughput in HS.

3.6.2 Deciding on a benchmark suite

It is easy to rule out TPCx-HS on the basis that it does not fit a big data use case for comparing file formats performance.

Both TPC-DS and TPCx-BB are viable options which cater to slightly different use cases. TPC-DS generates only structured data and has redundancy, whereas

TPCx-BB generates structured, semi-structured and unstructed data. DS has 99 queries in four different query categories to simulate different query approaches, whereas BB has just 20 queries in one category partially based on DS's queries.

The combination of the variety provided by TPC-DS and much better documentation has lead us to chose TPC-DS. Given our selection we will break down the TPC-DS specification and make decisions on our implementation of it in chapter 5.

Chapter 4

Framework

This chapter gives an overview of all the software packages we used together to form our framework, from setting up our infrastructure to displaying the experiment's final results. We introduce these software packages as components in the larger environment. Our goal was to choose certain software packages and leverage their functionalities when used together to result in an environment containing an entire cluster of virtual machines running on the cloud, all setup programatically according to our specification. The framework is extensible which can be adapted to fit different needs.

We leverage the Google Cloud Platform cloud computing provider and use Terraform, an infrastructure manager, to communicate with Google's API on our behalf in order to programatically build the hardware infrastructure in the cloud needed to form our cluster of computers. The Nix package manager is then used on each of the worker nodes. Each node runs on the NixOS Linux operating system. The packages required by our experiments are Hadoop and HDFS [20], Spark [21], Hive [16] and TPC-DS [17]. The following sections will give detailed explanations of how we setup and used each component in the framework. Our environment is developed in such a way as to be easily reproducible by other users easily, automating nearly every step of the building process. The big data processing experiments we chose to run are not automated. The Google Cloud Compute Engine service enforces quotas on the power and size of some of its resources. The free trial we used has limited quotas, namely our virtual machines' resources in total cannot exceed 8 virtual CPUs and 10 TB of disk storage [12]. Thus, our experiment scale was as large as these limitations allowed us. With available funds, anyone using our framework could modify parts of the code to easily both scale up and scale out the cluster, allowing for larger scale big data workloads.

4.1 Cloud Computing with Google Cloud Platform

Google Cloud Platform (GCP) offers a wide variety of cloud computing solutions. This section will focus on only the components relevant to the project, as Google Cloud Platform offers a multitude of hardware and services. Explaining all of them is out of scope of this paper.

The resources used for this project were instances of the *Google Compute Engine* (GCE) service, which are virtual machine instances with attached disks, acting as our cluster. The machines are complete computers, with processors, memory and hard disks to be used as needed. A Google Compute Engine instance contains three components [7]:

- **Virtual machines**: are virtual representations of hardware. Different hardware configurations are available through GCE, with different level of hardware performance and price, allowing the user to choose the one most suitable for their computational needs. GCE supports different operating systems, including distributions of Linux, UNIX or Windows. A GCE instance is a resource that is part of a geographical zone, which is chosen at build time [7].
- **Persistent disks**: are network-based block storage devices, either hard disk or solid state disk. They are attached to one or more virtual machines and act as an additional storage device. They remain persistent when their corresponding virtual machines are destroyed, preserving data. These disks can be up to 64 terabytes in size [7]. The fact that they are external resources to the virtual machine does not affect our experiments. They are used only to generate mock data on, before moving it into HDFS.
- **Networks**: are the ways which virtual machines can communicate with the world, including the persistent disk, which is hosted separately. Network bandwidth is given to virtual machines according to the number of CPU cores they have. The higher the number of CPU cores, the greater the bandwidth allocated to the corresponding virtual machine [7].

In an academic project such as this one, where a lot of learning was involved, we took in consideration all of GCP's benefits, along with resource reusability and ease of infrastructure management when developing it. We chose to use GCP, as the abstraction of hardware infrastructure that it provides would allow us to develop, test and run our experiments in Google's cloud without needing much hardware expertise or any financial upfront costs, as they offer a limited 300 dollars in their free trial package, which was the most out of any cloud computing providers offered.

We used Terraform [3] to setup our infrastructure programatically, as explained in the following section .

4.2 Setting up the Infrastructure with Terraform

This section explains our Terraform codebase, containing code snippets of the most important parts, along with their explanations. Some parts will be directly tied to Nix expressions, which are explained in greater detail in Section 4.3. This section will focus on the the code related to Terraform and the overall project artifact files.

Our entire project's codebase can be divided into three types of components: Terraform files with the *.tf* extension, Nix expression files with the *.nix* expression and the rest being artifact text files containing various information.

The most important artifact file is *account.json*, which contains service account keys corresponding to the Google Cloud Platform account that owns the entire project. This file has sensitive information, including the project's id, private key, authentication token etc. This file is necessary for Terraform to have access to, as it is the authentication mechanism to grant it administrator priviledges, allowing it to communicate with the GCP API on our behalf and create, destroy or make changes to any of our cloud resources. After making an account, creating our project, generating and downloading the account private key file, very little use of the GCP web portal was needed, as Terraform code handled the resources for us programatically, allowing us to have a better overview of our infrastructure and control over our development process.

Our project contains three Terraform files: main.tf, master.tf and worker.tf. Main.tf contains code relating to the overall GCP project, while master.tf and worker.tf have code related to the type of virtual machine instance it will be deployed to.

Main Terraform file

This subsection will explain the code found in the main.tf file, necessary for the foundation of the GCP project.

```
variable "env" {
  description = "Deployment environment name, used to prefix resource names."
  default     = "dev"
}
```

The variable "env" is an input parameter, to be used in customizing the other resource's attribute through interpolation. We chose the simple name "dev", but it can be anything.

The primary construct of the Terraform language is the resource, and their behavior rely on their corresponding types, which in turn are defined by the cloud *provider*. Providers offer different types of resources, each with their own definitions. These definitions include what arguments they accept, attributes they export and how to interact with them via the provider's API. In this case, we made use of the Google Cloud Platform provider, which needed to be configured. The following code snippet configures the GCP project and adds the secret key as a resource.

```
provider "google" {
  project = "symmetric-stage-242608"
  region  = "europe-west1"
}

resource "tls_private_key" "root_key" {
  algorithm = "RSA"
}

resource "local_file" "private_key" {
  filename = "${path.module}/SECRET_private_key"
  content  = tls_private_key.root_key.private_key_pem

}
```

The provider has a name, given in the block header, signifying the provider to be used. It is configured through its body, which contains attributes relating to its corresponding project, which must be created manually through the GCP portal, along with the region it resides in. Terraform uses this code block to then initialize the provider. The user, upon running the *terraform init* command, will cause Terraform to initialize the provider, which then downloads and installs the plugin corresponding to the chosen provider.

The second code block, *tls_private_key* will generate a secure private key resource, using the RSA algorithm to encrypt it. *root_key* signifies the resource's name.

The following resource, *local_file*, will generate a file in the given path with the newly generated private key as its content. The actual content is an attribute called *private_key_pem* of a resource of type *tls_private_key* named *root_key* Here, *$path.module* is interpolated inside the *filename* attribute in order to dynamically create the path to the local project's root directory at compile time and put the secret private key file there.

The operating system we chose to run on our virtual machines was the NixOS Linux distribution [5], a purely functional operating system with a purely functional package manager, Nix. Since GCP does not provide the option of setting up the Google compute instances with NixOS, we define the following custom resource with the its installation image for Terraform to download manually.

```
resource "google_compute_image" "nixos_1809" {
  name = "nixos-1809"

  raw_disk {
    source =
    "https://storage.googleapis.com/nixos-cloud-images
    /nixos-image-18.09.1228.a4c4cbb613c-x86_64-linux.raw.tar.gz"
  }
}
```

Terraform will ensure that NixOS version 18.09 will be downloaded and installed as the operating system of our virtual machines.

The following resource declares the firewall within the Google compute instances. Access between GCE instances is controlled by each of their firewalls.

```
resource "google_compute_firewall" "default" {
  name    = "${var.env}-default-allow-ssh-and-spark"
  network = "default"

  allow {
    protocol = "tcp"
    ports    = [22, 7077, 8042, 8044, 8080, 9000, 50070]
  }
}
```

Network traffic to or from these instances must be explicitly allowed by the firewall via creating its rules. The name resource contains the variable name, *env*, interpolated within its full string name. *allow* contains the rule that, using the TCP protocol, the specified ports should be allowed as permitted connections. In our case, these ports are being used by various software packages we use, such as Hadoop, Spark etc.

The following resource declares a persistent disk, acting as an external data storage device that will be attached to the master node of the cluster.

```
resource "google_compute_disk" "data_storage" {
  name                       = "${var.env}-test-disk"
  type                       = "pd-standard"
  size                       = 1000
  zone                      = "europe-west1-b"
  physical_block_size_bytes = 4096

  lifecycle {
    prevent_destroy = true
  }
}
```

The type *pd-standard* signifies that is a hard drive, as opposed to a solid state disk. We chose this option as it is the cheaper option in GCP. The size is measured in gigabytes. The size also affects the price. When developing, we chose smaller sizes for testing purposes and to save money, as it was easy to change to a larger size later on. The benefit of such a persistent disk is its ability to be attached and detached from different virtual machines. The *prevent_destroy* attribute will prevent this resource from being destroyed when running the *terraform destroy* command on the virtual machines. This allows the data stored to not be deleted. This proved useful in our project, as we were experimenting with different virtual machine configurations, while keeping the data intact. The *physical_block_size* attribute declares the block size of the disk.

```
data "template_file" "configuration_json" {
  template = <<EOF
{
  "master_ip": "${google_compute_instance.master.network_interface.0.network_ip}",
  "worker_ips": ${jsonencode(google_compute_instance.worker.*.network_interface
  .0.network_ip)}
}
EOF
}
```

template_file will create a template file in encoded in the JSON file format, named "configuration_json" containing all the nodes' internal IP addresses. The string values are retrieved from their respective node type configuration files after they have been built, namely master.tf and worker.tf.

```
locals {
  public_ip = google_compute_instance.master.network_interface.0
  .access_config.0.nat_ip
}

resource "local_file" "public_ip" {
  filename = "${path.module}/OUTPUT_public_ip"
  content = local.public_ip
}

output "public_ip" {
  value = local.public_ip
}
```

locals declares local variables, to be used only within the current Terraform file. In this case, *nat_ip* is the external, public IP address of the node. The rest of the code will generate a file containing the external, public IP address of the master node and put it in the project's root directory at build time.

Master and worker Terraform files

This subsection covers the code within master.tf and worker.tf. As their name suggests, their roles are deploying their respective virtual machine instance types. In our project's cluster, there is only one master node to multiple worker nodes.

The following code snippet declares the GCE instance's configuration corresponding to the cluster's master node.

```
resource "google_compute_instance" "master" {
  name         = "${var.env}-gcespark-master"
  machine_type = "n1-standard-2"
  zone         = "europe-west1-b"

  boot_disk {
    initialize_params {
      image = google_compute_image.nixos_1809.self_link
      size  = 40
    }
  }

  network_interface {
    network = "default"
    access_config {
      nat_ip = google_compute_address.gcespark.address
    }
  }

  metadata = {
    sshKeys = "root:${tls_private_key.root_key.public_key_openssh}"
  }
  connection {
    user        = "root"
    host        = self.network_interface.0.access_config.0.nat_ip
    private_key = tls_private_key.root_key.private_key_pem
  }

  provisioner "remote-exec" {
    inline = [
      "nix-channel --remove nixos",
      "nix-channel --add https://nixos.org/channels/nixos-19.03 nixos",
      "nix-channel --update"
    ]
  }
}
```

The *machine_type* signifies the type of virtual machine to be deployed. Several different types exist on GCP, with different strengths, and we chose a standard one with two CPU cores and 7.5 GB memory due to its low price and decent performance, especially for the developing and testing phase of the project which did not demand a lot of computing power. When needing more power, we chose to scale out the cluster, by adding more worker nodes with this type 2 machine hardware, rather than choose machines of type 4, with twice the power and memory but more costly. However, by using Terraform to manage the cloud infrastructure, changing the machine types is trivial, with only minimal code changes required before rebuilding. If needed, much more powerful virtual machine instances can be built at a moments notice and set up programatically via Terraform code.

The *boot_disk* attribute declares the main storage device of the machine. The persistent disk, which was previously defined in main.tf, is attached to the master node's virtual machine as a boot disk, with a size of 40 GB. This is the master node's main storage device, storing the operating system and various installed packages, but not our testing data, thus not a lot of storage space was required. This helped us keep costs down by only creating hardware that fits our computational needs and avoid overpaying. We needed this separate disk in order to generate our mock data locally before moving it to the cluster's HDFS. This is due to TPC-DS (explained in section 3.6.1) needing a local disk to generate data in.

The *network_interface* attribute specifies the network to be attached to the virtual machine. In our case, the default option with the machine's external public IP is sufficient. *nat_ip* within the *access_config* attribute will be bound to this value at build time. This allows this machine to be accessible from anywhere on the internet.

"Metadata" contains the SSH keys to be saved as key value pairs and made available in the virtual machine instance. This is necessary because they will not be stored in the Google Cloud, so it is important to attach them to the instance. The *metadata.sshKeys* variable will contain a mapping of the users to public keys. Those keys will be copied to the user's home directory: /.ssh/authorized_keys file/. The SSH server looks it up on connection attempt then verifies the key against the client's signature. *connection* will then allow the user *root* to be accessed remotely and securely, due to the private keys bound to the connection. We use this to connect to the any of the nodes remotely via SSH from our personal laptops.

The *remote_exec* provisioner runs the specified script on the virtual machine it corresponds to after its creation. In our case, Bash commands are executed in order to update the package manager's channel to the newer one.

```
resource "null_resource" "deploy_master" {
  count = 1
  triggers = {
    instance = google_compute_instance.master.id
    always   = uuid()
  }

  connection {
    user        = "root"
    host        = google_compute_instance.master.network_interface
    .0.access_config.0.nat_ip
    private_key = tls_private_key.root_key.private_key_pem
  }

  provisioner "file" {
    source      = "./nixos/"
    destination = "/etc/nixos/"
  }

  provisioner "file" {
    content     = "{ imports = [ profiles/master.nix ]; }"
    destination = "/etc/nixos/configuration.nix"
  }

  provisioner "file" {
    content     = data.template_file.configuration_json.rendered
    destination = "/etc/nixos/configuration.json"
  }

  provisioner "remote-exec" {
    inline = [
      "nixos-rebuild switch --show-trace"
    ]
  }
}
```

The rest of master.tf's code handles the actual deployment of the master node. The *null_resource* resource will actually move the specified files, present in the project directory, into the master node's file system. For example, the entire "nixos" folder, containing modules, packages and profiles will be moved into the master node's /etc/nixos directory. The appropriate *master.nix* profile configuration file is imported and then bounded to Nix's configuration profile. Applying these changes then require a rebuild of the configuration. This is done with the *nixos-rebuild switch –show-trace* command.

The worker.tf file has a lot of the same code, with slight differences.

```
resource "google_compute_instance" "worker" {
  count        = 4
  name         = "${var.env}-gcespark-worker-${count.index}"
  machine_type = "n1-standard-2"
  zone         = "europe-west1-b"


  ...
  resource "null_resource" "deploy_master" {
  count = 4
  triggers = {
    instance = google_compute_instance.master.id
    always   = uuid()
  }
  ...
```

For example, an additional *count* key value pairs are present, signifying the amount of clones of this configuration of virtual machine to be created and deployments of files to be executed on. Both count values need to be equal, as all files to be deployed are needed on all of the cluster's nodes.

These lines of Terraform code corresponding to our configuration and desired infrastructure are an important part of the software framework we designed. It handled our cluster's infrastructure, building it up from scratch many times on command, as we kept developing it and making changes. The declarative programming paradigm proved to be crucial when dealing with a lot of trial and error, saving us lots of time and effort throughout the process.

After the infrastructure is built, the next step is to setup the needed packages and configuration files automatically through the Nix package manager. The next section will explain the code within our Nix expressions.

4.3 Package Management with Nix

This section explains the most important parts of the Nix expressions we used in setting up our packages and configuration files in the cluster. Each node in the cluster runs the NixOS operating system [5], which contains the Nix package manager [4]. Thus, the Nix expressions are responsible for setting up the entire system of any given machine, not just install packages.

In order to execute large-scale queries, we use Spark, a big data processing engine that supports querying large datasets. Moreover, it works in a distributed fashion, using the master/worker architecture [21]. We also use Hadoop to store our mock data. It has its own Map Reduce distributed data processing engine, however we do not make use of it. The distributed version of Hadoop is named Hadoop Distributed File System (HDFS) and is used to distribute large data sets across computer clusters. This means that the HDFS storage space is that of the sum of all the cluster nodes' storage devices [20]. We set up our HDFS cluster to use all the worker nodes' disks as our distributed data store. We also installed and set up Hive. It is a Hadoop manager, compiling queries written in its specific HiveQL language into MapReduce jobs to be executed by Hadoop [16]. However, technical issues prevented us from setting up Hive to process our test queries in a distributed fashion. Thus, only Spark was used for this project. TPC-DS [17] is the software that we setup and use in order to generate mock data and queries, explained in subsection 3.6.1.

There are several Nix files within our project with their specific roles. Their functionality will be explained in this section, but for brevity, will not go into detail regarding the Nix expression language syntax. The following code snippet represents the directory tree containing the Nix files from our project.

```
/nixos/modules/
       |--hadoop_cluster.nix
/nixos/packages
       |--hadoop.nix
       |--tpcds.nix
/nixos/profiles
       |--common.nix
       |--master.nix
       |--worker.nix
/nixpkgs-pinned.nix
/shell.nix
```

The first step when building the infrastructure from scratch is done through the

Nix shell. The project files and the Nix package manager are on a local machine, such as a laptop, which allows us to start a Nix shell based on the configuration based off of a Nix expression. This Nix expression is named *shell.nix* and is used by the *nix-shell* command to build the dependencies of the derivation then start the Nix shell. In this shell, the environment variables are defined by the shell.nix derivation.

```
{pkgs ? import nixos/nixpkgs-pinned.nix {}, ...} :
let
  terraform = (pkgs.terraform_0_12.withPlugins (p: with p; [
    p.null local template tls google
  ]));
in
pkgs.stdenv.mkDerivation {
  name = "gcespark-env";
  buildInputs = [ terraform ];
  shellHook = ''
    [ -z $GOOGLE_CREDENTIALS ] && export GOOGLE_CREDENTIALS=account.json
    terraform init
  '';
}
```

The *let* expression gives names to values. *pkgs.terraform_0_12.withPlugins* is a function which takes another function that takes all plugins and returns a list of them to load. In this case, only few Terraform plugins are necessary. A derivation is made from these plugins bounded to the name "Terraform". The commands within *shellHook* are executed when the Nix shell is entered. "GOOGLE_CREDENTIALS=account.json" is an example of one of the environment varaibles available in the Nix shell. By adding it to the shellHook, the user does not have to add it manually when running Terraform commands. *terraform init* initializes the Terraform plan. Thus, the user, once in this Nix shell, must only run *terraform apply* in order to execute the build scripts that builds the entire infrastructure in GCP and sets up every package and configuration as specified in the Nix expressions and Terraform files. *pkgs* is the set of Nix packages, imported from the *nixpkgs-pinned.nix* expression:

```
    { pkgs ? import <nixpkgs> {}, ...} :
let
  pinned_pkgs_path = pkgs.fetchFromGitHub {
    owner = "NixOS";
    repo = "nixpkgs";
    rev = "8669561bde00b4039cda2b662f9f726db8385069";
    sha256 = "157a5h1vcfj892b20c90n7i6rfr5k61242ylgz6i21m8sbcxfry6";
  };
in
  import pinned_pkgs_path {}
```

Nixpkgs-pinned.nix is used to pin packages to a certain Nix packages (nixpkgs) repository commit with the id "8669561bde00b4039cda2b662f9f726db8385069". This ensures that the exact files from the exact repository commit will be retrieved every time they are needed. The list of available Nix packages and their paths are retrieved, which are then bounded as the variable "pkgs" and available in the other Nix expressions to be used.

The next step for Nix is to build the full system configuration. This is done by using *modules* containing Nix expressions. In our case, the *hadoop_cluster.nix* is responsible for our systems' configurations. Options are declared in the module to be given values by other modules, which in turn are other Nix expressions [5]. Our node cluster has two types of nodes: one master node and multiple worker nodes. This corresponds to the master/worker architecture adopted by the Hadoop Distributed File System, further explained in Tom White's "Hadoop: The Definitive Guide" [20]. This Nix expression handles the configuration for both types of nodes, setting up different system options to enable different deployments of package configurations, based on what type of node it is.

```
    { config, pkgs, lib, ... }:
let
  cfg = config.services.hadoopCluster;
in {
  options.services.hadoopCluster = with lib.types; {
    enable = lib.mkEnableOption "Hadoop node";

    master = lib.mkOption {
      type = bool;
      description = "Whether it is a master node";
    };

    master_ip = lib.mkOption {
      type = string;
      description = "IP address of master node";
    };

    worker_ips = lib.mkOption {
      type = listOf string;
      default = [];
      description = "List of IP addresses of worker nodes";
    };
  };
```

The *config* function argument, which contains the complete system configuration, is used to access configuration values from within other modules's code. Inside the *let* expression, the *hadoopCluster* system service is bound to the *cfg* value to be used in the body of the Nix expression to improve code readability. The *hadoopCluster* system service is then defined with several options. These options have variables with different types declared, and are to be used in *config*. These options will be interpreted as configurable NixOS options. For example, *master* will act as a switch with "true" or "false" values. When defining the master node's profile as a Nix expression (master.nix), inside the body the following line will be added to signify it is the master node: services.hadoopCluster.master = true; while the worker node's profile Nix expression will contain the same configuration with the "false" value bound to it. The other data structures are *master_ip*, which will be used to bound it to the master node's IP address. As there are multiple worker nodes, *worker_ips* is declared as a list of elements, each element being a string containing one worker IP address. The flexibility of the structure allowed us to easily scale out, adding multiple worker nodes without needing to change any parts of the Nix code.

The following code snippet shows most of the remaining *hadoop_cluster.nix* expression.

```
config = lib.mkIf cfg.enable {
    environment.systemPackages = [ config.services.hadoop.package ];

    users.groups.hadoop = {};
    users.users.hadoop = {
      group = "hadoop";
      createHome = true;
      home = "/home/hadoop";
    };

    system.activationScripts = {
      hadoopGroupRWX = {
        text = "chmod -R g+rwx ${config.users.users.hadoop.home}";
        deps = [];
      };

      tmpRWX = {
        text = "mkdir -p /tmp/hive && chmod -R a+rwx /tmp/hive";
        deps = [];
      };
    };
    services.hadoop = {
      hdfs.namenode.enabled = cfg.master;
      hdfs.datanode.enabled = !cfg.master;
      yarn.nodemanager.enabled = cfg.master;
      yarn.resourcemanager.enabled = cfg.master;
      coreSite = {
        "fs.defaultFS" = "hdfs://${cfg.master_ip}:9000";
      };
      ...
```

A configuration is declared to be activated only if the *hadoopCluster* service is enabled. The Hadoop package is added to the system's packages, which causes Hadoop to be installed on any machines with this configuration enabled. This is a pre-made module from the *Nix packages* repository in order to easily setup Hadoop in a distributed fashion, which forms the Hadoop Distributed File System (HDFS) as defined in [20], which has specific requirements when it comes to its configuration within the system it operates in. NixOS modules are merged on system activation to form the whole system configuration. This includes creating a system user group, specific Linux user and home directory, giving administrator permissions to directories which Hadoop needs to access. Scripts within *system.activationScripts* will be executed after every package is installed and the service started. By being able to interpolate variables into string values, writing such scrips is made easy. For example, the Hadoop home folder's path will be bound to "config.users.users.hadoop.home" when running the Nix expression, which is not known beforehand and cannot be hard-coded.

Other Hadoop-specific configuration files and services are easily changeable through several values. *services.hadoop* denotes the services that it will launch, namely HDFS and Yarn. Different daemons run on different machines, depending if they are master or worker nodes. For example Hadoop master node daemon is called the *namenode*, while its worker nodes are *datanodes*. In the example code shown corresponding to the master node, the options defined previously are used as configuration attributes to be bounded to Hadoop configuration settings such as "hdfs.namenode.enabled = cfg.master". This setting will be set to "true" if the corresponding node is defined as a master node, and the opposite case is true if the node is defined as a worker (!cfg.master). The *coreSite* corresponds to the core-site.xml file that is part of Hadoop and needs configured. Normally this is done manually, but through Nix packages, it has to be configured programmatically, as shown above. Interpolation lets keys to be bounded with values that are dynamically decided at build time. In the case of "coreSite", "fs.defaultFS" will be bounded to the HDFS address, which is the master node's IP address. The *cfg.master_ip* value will be replaced with the actual IP address string at build time. IP addresses are retrieved from the "configuration.json" file, which is created in "main.tf". Several of these Hadoop configuration files are changed by Nix programmatically, which not shown for brevity. These include the HDFS specific configuration files: core-site.xml, yarn-site.xml, hdfs-site.xml and mapred-site.xml which HDFS needs [20]. Nix will create these XML configuration files and put them in the appropriate directories. The HDFS configurations are easily changeable through Nix code if needed, adding to the further flexibility offered by using Nix. The modified *hadoop.nix* package is imported at the end of this file:

```
package = import ../packages/hadoop.nix
{ inherit pkgs; inherit (cfg) worker_ips; };
```

It refers to the result of the *hadoop.nix* expression and it is then passed as an environment system variable in the resulting machine. This *package* variable is the one referenced in *environment.SystemPackages* above.

We needed to make a small adjustment to Hadoop package's code in order to prevent a warning message to appear every time a Hadoop command was executed. This is possible in Nix with few lines of code:

```
{ pkgs ? import <nixpkgs> {}, worker_ips? ["localhost"], ... }:
pkgs.hadoop_3_1.overrideAttrs (oldAttrs: {
  installPhase = builtins.replaceStrings ["HADOOP_PREFIX"]
  ["HADOOP_HOME"] oldAttrs.installPhase + ''

    echo "${pkgs.lib.concatStringsSep "\n" worker_ips}\n" > $out/etc/hadoop/workers
  '' ;
})
```

The hadoop.nix expression overrides the Hadoop package's configuration file and adds the worker node IP addresses in Hadoop's "workers" file. By using the *installPhase* function in Nix, it is possible execute some commands and override code at the package's install phase. In our case, two commands were needed: the string "HADOOP_PREFIX" should be replaced with "HADOOP_HOME", due to the latest version of Hadoop needing the name change of the system environment variable in order to not show a warning every time a Hadoop command is executed. "HADOOP_HOME" points to the path where Hadoop is installed on the system. The second task required was to automatically populate the "workers" configuration file with all of the worker nodes' IP addresses. Developing small changes and scrips like these reduce setup overhead when it comes to reproducing the whole system on other machines. Instead of needing to write instructions for another developer to follow in order to have the same configuration as us, the Nix expressions will handle all these changes automatically.

Another package we needed was TPC-DS (presented in section 3.6.1). It required creating a Nix derivation for, as there was no pre-made Nix package available from the official repository. Nix is versatile, allowing for custom derivations such as the following:

```
    { pkgs ? import <nixpkgs> {}, ... }:
pkgs.stdenv.mkDerivation rec {
  pname = "tpcds-kit";
  version = "unstable-2019-06-14";
  src = pkgs.fetchFromGitHub {
    owner = "gregrahn";
    repo = pname;
    rev = "9d01e73403c32d8e3d89a987d096b98cbfae3c62";
    sha256 = "0l1jn2k4n9cyvf3i4bjkirqpz77d42jv13yzwg34rwlzckrvybx5";
  };
  buildInputs = with pkgs;
  [ makeWrapper bison byacc (writeScriptBin "lex" "exec ${flex}/bin/flex $@")];
  hardeningDisable = [ "all" ];

  preBuild = ''
    cd tools
  '';
  installPhase = ''
    mkdir -p $out/bin
    cp -r dsdgen dsqgen distcomp mkheader checksum .ctags_updated $out/bin
    mkdir -p $out/share
    cp -r tpcds.idx ../query_templates $out/share
  '';
  postFixup = ''
    wrapProgram $out/bin/dsdgen --run "cd $out/share"
    wrapProgram $out/bin/dsqgen --add-flags "-directory query_templates"
    --run "cd $out/share"
  '';
}
```

It is a basic derivation that downloads the TPC-DS source from a GitHub repository, sets up the build inputs and then executes commands. The "buildInputs" function is bounded to the list of packages required to build TPC-DS. We discovered a small issue with the TPC-DS package, namely that attempted to call the Lex package, which was not available. The solution to this is using the "writeScriptBin"

Nix function to create a Lex binary that, when called by TPC-DS build process, simply uses Flex instead. Other useful functions such as "preBuild", "installPhase" and "postFixup" allow for shell commands to be injected in various phases of the TPC-DS package creation lifecycle. Thus, by setting up these commands to run automatically before, after or during the build process, the TPC-DS package will be properly setup for the end user. Learning how TPC-DS package works and what setup it requires involved lots of trying and testing, but the result is a reproducible script to be ran as many times as we need to rebuild the whole system.

The remaining parts of the Nix codebase involve the Nix *profiles*, stored in the directory with the same name.

```
/nixos/profiles
   |--common.nix
   |--master.nix
   |--worker.nix
```

Profiles are used to distinguish the difference between deployments for different node types. For example, configuration files and services defined in *common.nix* are to be deployed to all nodes, as to not have repeated code in both the *master.nix* and the *worker.nix* Nix expression files.

The following code snippet is part of *common.nix*:

```
{ config, pkgs, lib, ...}:
let
  configurationJson =
    if builtins.pathExists ../configuration.json then
      builtins.fromJSON (builtins.readFile ../configuration.json)
    else
      lib.warn "../configuration.json not found" {};
in{
....
```

The configuration JSON file is first created in *main.tf*. It contains the nodes' IP addresses. This file is then read by the *builtins.readFile* Nix function and bounded to the *configuationJson* variable.

```
    imports = [
    <nixpkgs/nixos/modules/virtualisation/google-compute-config.nix>
    ../modules/hadoop_cluster.nix
  ];
```

Two Nix modules are then imported: the one necessary for configuring the Google Compute Engine instance, which is premade, and the hadoop_cluster one modified by us. The *hadoop_cluster.nix* expression file is imported to the this profile because all the nodes will be part of the Hadoop cluster, master and workers, thus needing the common Hadoop specific configuration files and services running on them.

The rest of the code handles the common configurations:

```
config = {
    networking.firewall.enable = false;
    networking.hosts =
      builtins.listToAttrs (lib.imap0 (i: x: {
        name=x;
        value=[("worker" + builtins.toString i)];
        }) configurationJson.worker_ips)
        // {"${configurationJson.master_ip}" = ["master"];
        };


    users.groups.spark = {};
    users.users.spark = {
      group = "spark";
      createHome = true;
      home = "/home/spark";
    };

    services.hadoopCluster = {
      enable = true;
      master_ip = configurationJson.master_ip;
      worker_ips = configurationJson.worker_ips;
    };

  };
```

There is no need for the machines' builtin Linux firewall, they already have the firewall from GCP, thus we disable it. The system *hosts* file will be mapped to the master and worker nodes IP addresses, along with a unique name for each of them. An example resulting hosts file would look like:

```
0.0.0.0 worker0
0.0.0.1 worker1
0.0.0.2 master
```

The *spark* user and usergroup is created for Spark to use. The Hadoop cluster service is then declared to be enabled, and the *master_ip* and *worker_ips* values are bounded ot their respective IP addresses. These values will be usable in the *master.nix* and *worker.nix* Nix expressions for further configurations.

The following code snippets will explain the *master.nix* expression and its functionality:

```
    { config, pkgs, lib, ... } :
let
  pinnedPkgs = import ../nixpkgs-pinned.nix {};
  hive = fetchTarball {
    url =
    http://dk.mirrors.quenda.co/apache/hive/hive-3.1.2/apache-hive-3.1.2-bin.tar.gz;
    sha256 = "1g4y3378y2mwwlmk5hs1695ax154alnq648hn60zi81i83hbxy5q";
  };
  spark = pinnedPkgs.spark;
  tcpds = import ../packages/tpcds.nix { inherit pkgs; };
in
...
```

The actual packages are imported to be then deployed on the master node. The Nix package list from its repository is retrieved and bounded to *pinnedPkgs*. Spark is an existing Nixpkgs module which we did not modify at all, like we did with Hadoop, thus importing it via *pinnedPkgs.spark* is straightforward. There is no Hive Nixpkgs module, therefore we needed to invoke the *fetchTarball* Nix function in order to download its source archive. This function handles the extraction of the archive automatically and deletion once it is done extracting. TPC-DS is imported from the the Nix expression we defined previously.

```
imports = [ ./common.nix ];

  fileSystems."/data" = {
    device = "/dev/sdb";
    autoFormat = true;
    fsType = "ext4";
  };

  environment.systemPackages = [
    tcpds
    hive
    spark
  ];

  users.groups.hive = {};
  users.users.hive = {
    group = "hive";
    createHome = true;
    home = "/home/hive";
  };

  services.hadoopCluster.master = true;
```

The *common.nix* profile is then imported. This is done in both master and worker profiles. The separately created persistent disk in GCP is attached to the master node: "/dev/sdb" is the device the operating system sees, which will then be mounted in path "/data"/. We use this device to generate TPC-DS data on before moving it to HDFS.

The system environment packages are defined and added to the system path. The list bound to *environment.systemPackages* point to several packages, which at build time will point to their local paths where they are installed.

The "services.hadoopCluster.master = true;" line defines this node as being the master node within the Hadoop cluster configuration, thus triggering the deployment of services and configuration files needed by the Hadoop name node (master).

The Hive service is then defined, with its required configurations:

```
systemd.services.hive = {
    wantedBy = [ "multi-user.target" ];
    path = with pkgs; [ config.services.hadoop.package hive bash gawk procps which ];
    environment = {
      HIVE_HOME = hive;
      HADOOP_HOME = config.services.hadoop.package;
      HADOOP_HEAPSIZE = "2048";
    };
    serviceConfig.User = "hive";
    script = ''
      hdfs dfs -mkdir -p hdfs://master:9000/tmp/hive
      hdfs dfs -mkdir -p hdfs://master:9000/user/hive/warehouse
      cd
      schematool -initSchema -dbType derby || true
      hiveserver2\
        --hiveconf hive.metastore.schema.verification=false\
        --hiveconf hive.server2.enable.doAs=false\
        --hiveconf fs.defaultFS=hdfs://master:9000/\
        --hiveconf org.jpox.autoCreateSchema=true
    '';
    after = [ "hdfs-namenode.service" ];
  };
```

The "wantedBy = ["multi-user.target"]" line triggers the services to be launched. Hadoop, Hive and other packages are added to the path to be used by the Hive service. Hive required environment variables are declared such as *HIVE_HOME* and *HADOOP_HOME* easily through the *environment* configuration attribute. Shell commands to initialize the Hive service and have it be ready for use on system bootup are executed, saving us time when building the whole environment from scratch, thus not needing to manually start the Hive service and its related initializations every time. The "after = ["hdfs-namenode.service"];" line specifies after which service should this service start. In this case, Hive service should start after the Hdfs-namenode service has started.

```
systemd.services.spark-master = {
    wantedBy = [ "multi-user.target" ];
    environment = {
      SPARK_MASTER_HOST = config.services.hadoopCluster.master_ip;
      SPARK_LOG_DIR = "${config.users.users.spark.home}/logs";
      SPARK_NO_DAEMONIZE = "true";
    };
    serviceConfig.User = "spark";
    path = [ pkgs.procps ];
    script = ''
      ${spark}/lib/spark-2.4.3-bin-without-hadoop/sbin/start-master.sh
    '';
  };
```

Similar code is written to handle the Spark service and start it at system bootup by executing Spark's *start-master.sh* binary executable.

The *worker.nix* profile contains less code than the master, as its role is only to store files and execute commands got from the master. It is declared as a non-master, thus Nix knowing not to deploy master-specific services and files to it.

```
      { config, pkgs, lib, ... } :
let
  master_ip = config.services.hadoopCluster.master_ip;
  spark = (import ../nixpkgs-pinned.nix {}).spark;
in
{
  imports = [ ./common.nix ];

  services.hadoopCluster.master = false;
  systemd.services.spark-worker = {
    wantedBy = [ "multi-user.target" ];
    serviceConfig.User = "spark";
    path = [ pkgs.procps ];
    environment = {
      SPARK_MASTER_HOST = master_ip;
      SPARK_LOG_DIR = "${config.users.users.spark.home}/logs";
      SPARK_WORKER_DIR = "${config.users.users.spark.home}/work";
      SPARK_NO_DAEMONIZE = "true";
    };
    script = ''
      ${spark}/lib/spark-2.4.3-bin-without-hadoop/
      sbin/start-slave.sh spark://${master_ip}:7077
    '';
  };
}
```

Thus, the Nix language has enough expressive power to handle such specific package setups and various configurations as highlighted in this section. These Nix expressions are crucial in making the whole environment reproducible with as few manual steps as possible.

Chapter 5

TPC-DS Theory and Decisions

5.1 TPC-DS Partial Implementation

The specification and guidelines outlined by the TPC-DS specification explicitly details how a valid benchmark is meant to be performed, along with valid substitutions, alterations, and guidelines on how to validate configurations and operations that stray a little from the expected implementation and use.

Given the limitation of our resources, we have decided to implement a sub-set of the TPC-DS benchmark, using it more as a guideline. This of course invalidates our benchmarks in terms of successful TPC-DS benchmark results, however the results of our benchmarks are performed as fairly as possible adhering to the principles of TPC-DS and the benchmarking theory covered in this report to deliver trustworthy results, as detailed in Section 3.5, the information detailing TPC-DS in this section, and as summarised in subsection 5.1.2. Given that the tool created through this report can be easily scaled up, and the correct tools can be utilised as specified in the TPC-DS specification, future work can be done to enable successful TPC benchmarks as discussed in Future Work in Section 8.2.

The general implementation guidelines state that benchmarks using TPC-DS must use products that are available to users, are relevant, would actually be implemented and used by a significant amount of users, and queries must be executed from a SQL interface[14], which our project implementation abides by, as we are using Spark to submit our SQL like queries, onto Parquet and ORC files. Prohibited by the TPC-DS benchmark are specialised/customer systems that are not widely available, or designed specifically to optimise performance on the benchmark results. Our project is neither of these as we are using the latest public versions of our chosen software without any specialised configurations, however given the nature of infrastructure as code, the project could be manipulated into a prohibited implementation.

5.1.1 TPC-DS Setup

A summary of our setup is explained in Section 5.1.2 without all the TPC-DS theory, for a clearer understanding of all the decisions we have made.

The retail use case the benchmark imitates, can be broadly understood from the figure 5.1. The simulated business model uses five components which are sales channels (Store, WEB, Catalog), inventory and promotions.

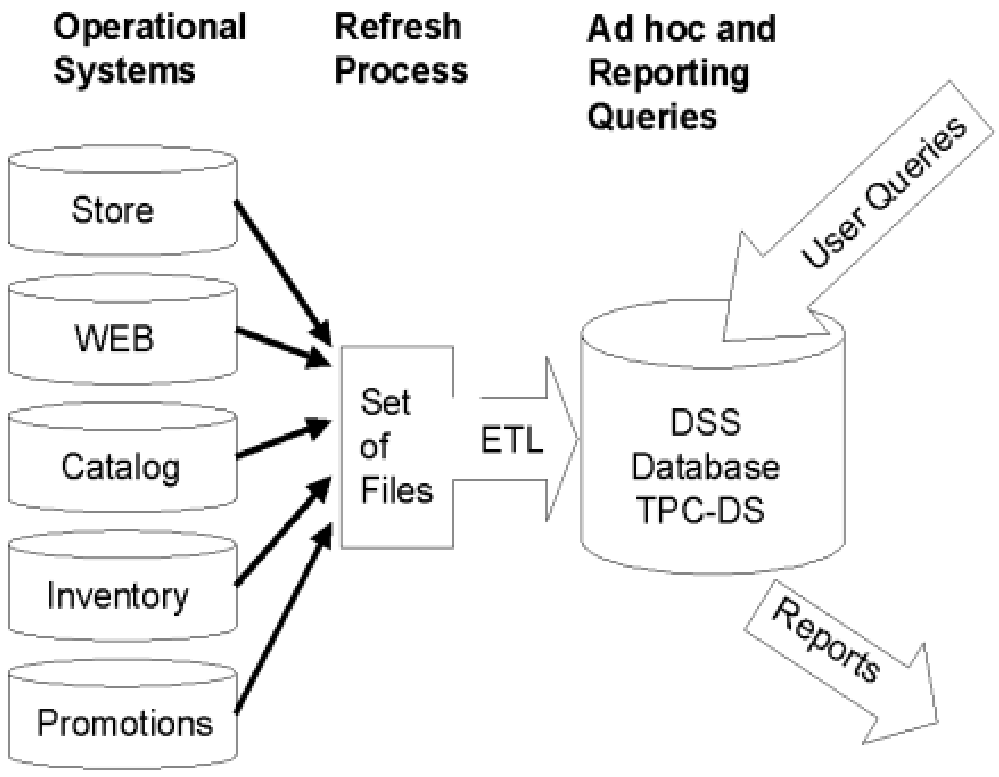

Figure 5.1: Components of the benchmark [14, p 16]

The components are composed of seven fact tables, which are linked to 17 dimension tables in a snowflake schema. The seven fact tables are: *store_sales*, *store_returns*, *catalog_sales*, *catalogue_returns*, *web_sales*, *web_returns* and *inventory*. All the fact tables are linked to the same 17 dimension tables through the dimensions tables surrogate key, so the fact tables are often joined and unioned to each other through the dimension tables.

Both the fact tables, and the dimension tables are generated as text files by a tool within the suite called dsdgen (DS - data - generator). It is typically called with a few parameters, such as directory to generate data into, delimiter to generate between the fields of data, a suffix for the file names, and a scale factor. The scale factor determines the total size of all the file combined, as one of the values seen in

Scale Factor	SF
1GB	1 (qualification database)
1TB	1000
3TB	3000
10TB	10000
30TB	30000
100TB	100000

Table 5.1: dsdgen scale factor [14, p. 39]

the table 5.1. The 1GB is not a valid scale factor for the benchmark, as it is merely used to validate a dataset if some modifications have been made to the data; TPC-DS specifics allowable modifications in its document specification, and also details guidelines on how to validate that data to be accepted as a valid benchmark using the 1GB scale factor. The values in the SF column in table 5.1, roughly translate to the number of GB that will be generated. An example execution of dsdgen executed independently is:

dsdgen -dir /data -scale 1 -verbose Y -delimiter "," -suffix .csv

As previously mentioned in Section 3.6.1, there are 99 queries which fit into four categories of query behaviour. It is explained in the TPC-DS specification [14] that the *catalog_sales* fact table is dedicated for the reporting queries, while the store and web fact tables use ad-hoc queries. The query distribution is: Reporting Class 41, Ad Hoc Class 59, Iterative Class 4, Data Mining Class 23. We have verified this by looking into each of the queries and searching for the *catalog_sales* fact table, which was found in exactly 41 queries (presumably all reporting queries): *2, 4, 5, 10, 14a, 14b, 15, 16, 17, 18, 20, 23a, 23b, 25, 26, 29, 32, 33, 35, 37, 38, 40, 49, 54, 56, 57, 58, 60, 64, 66, 69, 71, 72, 75, 76, 77, 78, 80, 87, 97 and 99*.

The TPC-DS specification does not provide a mapping of query number to query category, nor can a mapping be found online. An assumption must be made that the 41 queries involving *catalog_sales* must all be reporting queries in essence, but given the numbers of the query categorisation do not total to 99 (it totals to 127), then some of the queries must have multiple query behaviours defined, such as reporting also including data mining, and thus not much else can be ascertained about the other queries.

The reporting queries are defined as queries executed periodically, answering well known business questions. Ad-hoc queries are dynamic in nature, as they are constructed to answer specific and immediate questions about the business. The iterative queries are used to explore the data to find new relationships and trends within the data. Data mining queries are used to predict future trends and produce data content relationships.

We have chosen to model our use case after the *catalog_sales* fact table as its data makes up 40% of the entire dataset generated [14, p. 18], and it is likely that, most, if not all the queries involving it are reporting queries, which are guaranteed to handle big data sets, and return big data sets.

Of the 41 reporting queries, we have selected five of them to perform our benchmarking tests. We first wanted two relatively simple queries, without too many different operations being called to serve as a starting point.

The first query selected was number 15, as can be seen in Appendix B, line 438-452, as it selects a few fields across four tables, grouping and ordering the results. The business question the query seeks to answer, is defined in the TPC-DS specification as: *"Report the total catalog sales for customers in selected geographical regions or who made large purchases for a given year and quarter."* [14, p. 108]

38, as can be seen in Appendix B, line 461-490, was chosen next as it adds just a little more complexity, as it contains multiple selects which are combined with an intersect. The business question is: "*For the groups of customers living in the same state, having the same gender and marital status who have purchased from stores and from either the catalog or the web during a given year, display the following: • state, gender, marital status, count of customers • min, max, avg, count distinct of the customer's dependent count • min, max, avg, count distinct of the customer's employed dependent count • min, max, avg, count distinct of the customer's dependents in college count*" [14, p. 114]

Our next query is 56, as can be seen in Appendix B, line 639-703, and it has been chosen as it combines all the operations from the previous two queries, into a much bigger query spanning many more tables; it does however use union instead of intersect. The business question : *"Compute the monthly sales amount for a specific month in a specific year, for items with three specific colors across all sales channels. Only consider sales of customers residing in a specific time zone. Group sales by item and sort output by sales amount. "* [14, p. 119]

Query 49, as can be seen in Appendix B, line 499-630, was chosen next as it add more complexity on top of 56 by adding functions into the query. The business question: *"Report the worst return ratios (sales to returns) of all items for each channel by quantity and currency sorted by ratio. Quantity ratio is defined as total number of sales to total number of returns. Currency ratio is defined as sum of return amount to sum of net paid."* [14, p. 118]

Finally, the last query is 60, as can be seen in Appendix B, line 712-773, It selects across the three *sales* fact tables, and uses a sum function. The business question: *"What is the monthly sales amount for a specific month in a specific year, for items in a specific category, purchased by customers residing in a specific time zone. Group sales by item and sort output by sales amount."* [14, p. 120]

Many queries have been left out which could have been good additions, and even queries in the other query classes, like data mining and ad-hoc contain very interesting queries. We think these queries should be explored in future work.

To completely compare Parquet and ORC in certainty, various query behaviours would need to be performed to get an extensive understanding of which behaviours each file format excels in, rather than which operations and degree of joins/unions, number of columns selected, etc.

The 99 queries are in "Query Templates". A query template is a language independent format, that is translated into a language specific format by a TPC-DS tool called dsqgen (DS - query - generator), as explained previously in Section 3.5.4. The dsqgen tool is executed by the driver, which was explained in Section 3.6.1, and then the translated templates can be executed in the query stream. Our implementation in this report does not use the driver, as we are partially implementing the TPC-DS specification, and therefore we would be unable to successfully pass the validation measures executed by the driver. We therefore must be able to translate the templates, into PySpark code. This is actually accepted by the TPC-DS specification: *"The application of all minor query modifications must result in queries that have equivalent ISO SQL semantic behavior as the queries generated from the TPC-supplied query templates"*[14, p. 46]. The specification continues to explain in [14, p. 44] Chapter *4.2.3 "Minor Query Modifications"*, what constitutes valid modification of query templates that do not mirror the original functionality.

We are not modifying the five queries we have selected, however they are in a format we can not use. In order to execute the queries, we have found a Git-Hub project that offers the same 99 queries as Hive SQL statements, instead of the TPC-DS templates. This posed a new challenge that we would have to execute pure SQL statements that are intended to run on SQL tables; and given that the data generated by dsdgen does not contain headers, and we don't have tables to load the data into, we needed to simulate SQL tables in PySpark.

The schema for the *catalog_sales* can be seen in Figure 5.2, and the catalog_sales column definitions can be seen in Table 5.2. The schema for each fact table is relatively similar, with the difference being the selection of dimension tables e.g. all link to the *Date_Dim* dimension, only the *'Returns'* fact tables link to the *'Reason'* dimension (*Web_Returns*, *Catalog_Returns*, *Store_Returns*).

The column definitions table for *Catalog_Sales* seen here 5.2, has five columns and 34 rows. The first column is the column name, then the datatype for the data, then whether or not the column allows nullability, in which they all implicitly are nullable, except for the rows that specify 'N' in the 'NULLs' field, the next column indicates if a column is a primary key or not, and lastly Foreign keys show the surrogate keys to the other dimensions. The value *'decimal(7,2)'* in the datatype field means that it is a decimal value, that holds a maximum of seven numbers, with a precision of two numbers on the right hand side of the decimal point, e.g. 12345.67.

Column	Datatype	NULLs	Primary Key	Foreign Key
cs_sold_date_sk	identifier			d_date_sk
cs_sold_time_sk	identifier			t_time_sk
cs_ship_date_sk	identifier			d_date_sk
cs_bill_customer_sk	identifier			c_customer_sk
cs_bill_cdemo_sk	identifier			cd_demo_sk
cs_bill_hdemo_sk	identifier			hd_demo_sk
cs_bill_addr_sk	identifier			ca_address_sk
cs_ship_customer_sk	identifier			c_customer_sk
cs_ship_cdemo_sk	identifier			cd_demo_sk
cs_ship_hdemo_sk	identifier			hd_demo_sk
cs_ship_addr_sk	identifier			ca_address_sk
cs_call_center_sk	identifier			cc_call_center_sk
cs_catalog_page_sk	identifier			cp_catalog_page_sk
cs_ship_mode_sk	identifier			sm_ship_mode_sk
cs_warehouse_sk	identifier			sm_ship_mode_sk
cs_item_sk (1)	identifier	N	Y	i_item_sk
cs_promo_sk	identifier			p_promo_sk
cs_order_number (2)	identifier	N	Y	
cs_quantity	integer			
cs_wholesale_cost	decimal(7,2)			
cs_list_price	decimal(7,2)			
cs_sales_price	decimal(7,2)			
cs_ext_discount_amt	decimal(7,2)			
cs_ext_sales_price	decimal(7,2)			
cs_ext_wholesale_cost	decimal(7,2)			
cs_ext_list_price	decimal(7,2)			
cs_ext_tax	decimal(7,2)			
cs_coupon_amt	decimal(7,2)			
cs_ext_ship_cost	decimal(7,2)			
cs_net_paid	decimal(7,2)			
cs_net_paid_inc_tax	decimal(7,2)			
cs_net_paid_inc_ship	decimal(7,2)			
cs_net_paid_inc_ship_tax	decimal(7,2)			
cs_net_profit	decimal(7,2)			

Table 5.2: catalog_sales column definitions [14, p. 24]

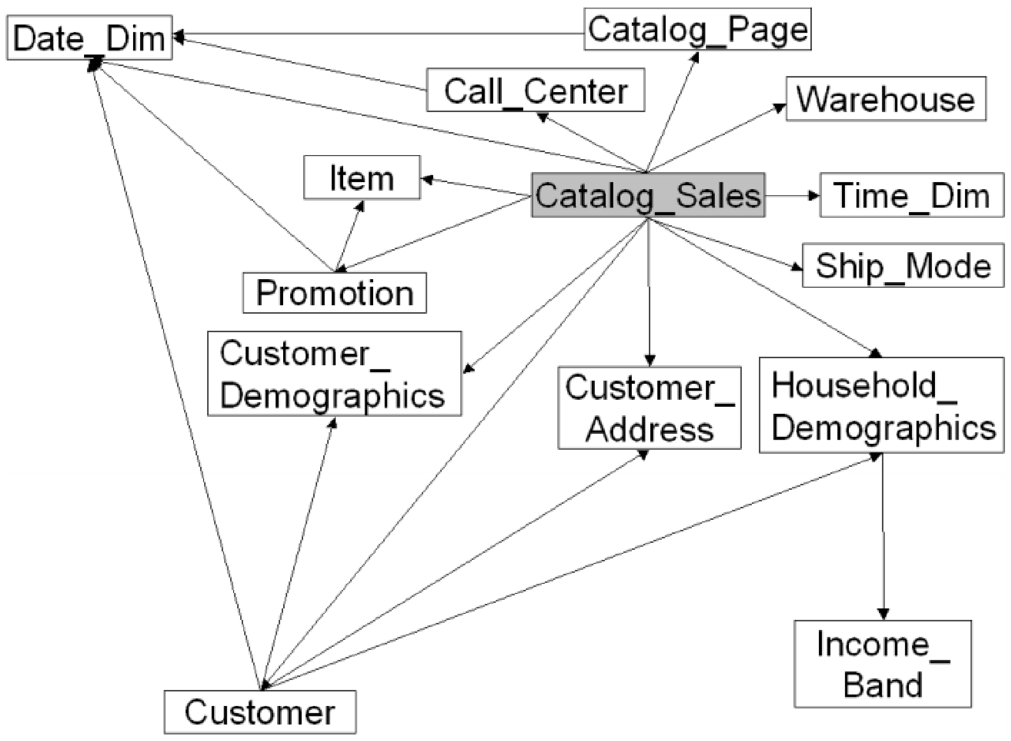

Figure 5.2: catalog_sales schema[14, p. 24]

Data Maintenance / Data Refresh / ETL

TPC-DS defines the process of ETL as something called *data maintenance* or *data refresh*. It is called data maintenance/data refresh as it lacks the "extract" part of ETL, as the data is generated; therefore it is focused on the "transformation" and "load" parts of ETL. A generated dataset is referred to as a refresh data set, and each refresh dataset must only have one query stream applied to it. In our case, that would mean we would need to generate the data twice, once to be transformed into Parquet and queried with our selection of queries, and then once again for ORC. However, we generate the data once using dsdgen, and create Parquet and ORC files from the generated CSV files, and then run two performance tests; one test for Parquet, then a separate run for ORC. Our chosen scale factor is 1. Scale factor 1 is only used by TPC-DS for verify data is being generated correctly and compliant. We have resorted to using this scale factor due to technical issues in generating and performing larger scale benchmarks due to free tier limitations on resources. As previously mentioned in section 4.1, our GCP account was limited by the free trial quotas, meaning that using a larger scale factor was out of our scope due to time resource and time constraints. However, given enough financial resources to upgrade the account, scaling up would be a trivial process, and large scale factor experiments would be possible.

The data can be loaded directly into the SUT as it is (in its normalised form), or it can be denormalised or mapped across multiple tables; however, a specialised loading tool may not be used, only generic loading processes may be used. In our case, we will be directly loading the file as they are into HDFS.

5.1.2 Summary of our process

We have decided not to use the driver process from TCP-DS because: We don't want to use all 99 queries which would cause TPC-DS validation check failures, and we are going to use an invalid scale factor (1).

Since we are not using the driver process, we are adapting the TPC-DS process. A bash call, will call dsdgen to generate data as a text file with a comma specified as the delimiter, and ".csv" specified as the suffix, meaning we generate CSV files which we can transform into Parquet and ORC respectively using Spark. The CSV files will be generated onto the local file system in the */data/* directory, transformed into Parquet/ORC, and then *-put* into HDFS, where a PyScript will perform the benchmark on them.

We selected five queries from the 99 queries available: 15, 38, 49, 56 and 60. We have chosen an invalid scale factor 1 due to GCP free tier limiting the size of our cluster to 2TB which is not sufficient for generating the first valid scale factor at 1TB, given we need an external disk and a local disk, each capable of holding the generated data, in addition the some space reserved by the master (it is worth noting that the 1TB scale factor generates a little more than 1TB, so it does not just fit nicely in our 2TB cap). Future experiments need to be carried out on a non-free tier to execute the first valid scale factor.

A PySpark script will run separately for both Parquet and ORC to adhere to the TPC-DS specification. The benchmark is run twice per SUT, so we will run the queries twice for Parquet and ORC. However given we are currently using a scale factor of 1, we are just executing the queries once per file format as the performance metrics aren't too interesting on such a small size of data.

Chapter 6

Benchmark Setup

To begin the Benchmarking process, the virtual machine was launched, the terminal was opened in the project folder, and the terminal commands to build and ssh into the GCP box were executed, as explained in Chapter 4, and shown in listing 1.

```
    Nix-shell
    terraform apply -auto-approve
    ssh -i SECRET_private_key root@<output ip here>
```

Listing 1: *GCP build and ssh bash commands*

Once in the GCP box, the data for the benchmark is generated from any location on the terminal using the command seen in listing 2.

```
    dsdgen -dir /data -scale 1 -verbose Y -delimiter "," -suffix .csv
```

Listing 2: *data generation using TPC-DS dsdgen*

As explained in subsection 5.1.2, the scale factor is set to 1, roughly translating into 1GB of data, and the data is written out as a text file without a header.

It is in the external disk that the CSV files are converted into Parquet and ORC files using a PySpark script that loads the CSVs into dataframes, and writes out Parquet and ORC as seen in listing 3. The PySpark script called *transform.py* is located in the */data/pythonscripts/* directory, and can be seen in Appendix A.

```
    df=spark.read.format("csv").load("/data/catalog_sales.csv")
df.write.parquet("/data/parquet_catalog_sales")
```

Listing 3: *Snippet showing a CSV to Parquet transformation*

To execute the *transform.py*, you must first navigate to the folder spark is contained in, and then using *spark-submit*, in conjunction with the path to the script, Spark executes the script. Example in listing 4.

```
cd /nix/store/s4f9b100j97s172s41av7szgfjcnmmbs-spark-2.4.3/
./bin/spark-submit /data/pythonscripts/transform.py
```

Listing 4: *Bash commands to execute PySpark scripts*

As this data is generated onto an external disk, for reasons explained in section 4.3, we must put the Parquet and ORC files into HDFS with these commands seen in listing 5.

```
hdfs dfs -put /data/parquet* hdfs://<ip>:9000/
hdfs dfs -put /data/orc* hdfs://<ip>:9000/
```

Listing 5: *HDFS put commands to move files into HDFS*

With the files on HDFS, spark can execute the benchmark script locally. The benchmark script is called *benchmark.py* and is located in */data/pythonscripts/* with the other PySpark script. It can be seen in Appendix B. The script is designed to be run twice, once for benchmarking Parquet, and again for ORC. The files are read into a dataframe, and then each column within the dataframe is selected and renamed with an alias. This is required as the data is generated without a header, so the columns are named *_col0, _col1...*; the columns must be named correctly due to use using SQL queries that match the name of the columns seen in the specification. Listing 6 shows an example of creating a temporary table from a ORC file called *orc_catalog_sales*.

```
df = spark.read.orc("hdfs://<ip>:9000/orc_catalog_sales")
df = df.select( col("_c0").alias("cs_sold_date_sk")...
df.registerTempTable("catalog_sales")
```

Listing 6: *Snippet for creating temporary tables*

With the temporary tables created, the SQL queries can execute. As the script is being run twice, the tables are named neutrally, e.g. *catalog_sales*, and not *parquet_catalog_sales*, allowing the queries to be run unmodified. In the current verison of the script, only time is recorded for each query, but that can of course be extended in future work. An example of a query being run can be seen in listing 7. The sql query is removed from this listing in order to save space; it however can be seen in Appendix B on lines 438 to 452.

```
    query_start_time = time.time()
    query1 = sqlContext.sql("""SELECT...""")
    query1.show()
    query_end_time = time.time()
    query_elapsed_time1 = (query_end_time - query_start_time)
```

Listing 7: *Snippet for timing query1*

Upon completion of all five queries, the execution times are returned to the main function as a list, and conveniently printed as the last print to make clear the execution times of the benchmark.

Chapter 7

Performance Evaluation

Given the tool created in this project was only executed on a dsdgen scale factor of 1, we can hardly give a performance evaluation of Parquet and ORC based on such a small size of data. However the tests were still run for the purpose of proving it does work, and all it requires is the appropriate resources to run on a sufficiently large dsdgen scale factor.

Query 1 (TPC-DS query 15)

Query 1 can be seen in Appendix B on lines 438 to 452.

Parquet	ORC
9.150	9.620

Table 7.1: Query 1 results

Query 2 (TPC-DS query 38)

Query 2 can be seen in Appendix B on lines 461 to 490.

Parquet	ORC
23.356	23.951

Table 7.2: Query 2 results

Query 3 (TPC-DS query 49)

Query 3 can be seen in Appendix B on lines 499 to 630.

Parquet	ORC
12.309	11.902

Table 7.3: Query 3 results

Query 4 (TPC-DS query 56)

Query 4 can be seen in Appendix B on lines 639 to 703.

Parquet	ORC
10.423	10.257

Table 7.4: Query 4 results

Query 5 (TPC-DS query 60)

Query 5 can be seen in Appendix B on lines 712 to 773.

Parquet	ORC
10.728	9.854

Table 7.5: Query 5 results

7.1 Result Analysis

While the dataset is not big enough to gain a deep understanding of the performance of the file formats, at least one interesting observation can be made on query 2. Query 2 has 30 lines of code, which is exactly in between query 1 with which has 16 lines of code, and query 5 with 63 lines of code. What makes query 3 unique from the other four queries is the use of DISTINCT and INTERSECT. This calls for the addition of some new queries including them to explore the impact of that statement/operator on the performance of Parquet and ORC.

The increased time on query 3 is likely due to the size of the query. Query 3 is 127 lines of code, and contains many functions.

Chapter 8

Conclusion

8.1 Discussion

This project was deceptively difficult. Our intention starting out on this project was to gain a deep understanding of Parquet and ORC query performance given various operators and functions, across different configurations of spark clusters (e.g. various sizes of dsdgen data scale factors, various worker nodes, understanding of the impact of number of CPU cores and memory, ect). However the project became less about Parquet and ORC, and more about system configuration, automatic scalable frameworks and benchmarking in general. This is likely due to us vowing from the previous semester, to put much more emphasis on theory and literature when conducting our next project.

Quickly into the project we had an environment in GCP, ready with TPC-DS and Spark configured and running. However we took our time to explore the theory of Terraform, Nix, benchmarking and TPC-DS without testing the system on more than a scale factor of 1 with a single query to check it worked.

We took for granted that Spark was operating fine, and pursued a sub goal of setting up Map Reduce in order to use Hive and Impala. We managed to get Hive to work but only locally, meaning on the master node. Hive's metastore was successfully exported to a separate PostgreSQL database. This was an important step in setting up Impala, as it needed an external metastore, since it doesn't have its own. More knowledge was needed to understand all the services running to make the MapReduce engine work accross a cluster. This was a big drawback, causing us to spend too much time debugging it without reaching a working state.

Parallel to figuring out the configuration to enable Map Reduce, we also wanted to explore what a valid Parquet/ORC performance test would be, as we had stated in our previous project that we had little confidence in our results, due to it not being backed by any theory/literature. A considerable amount of time was used in understanding the tools the TPC offered, and how to use the three relevant ones covered in this report, however we did not actually use those tools as the TPC intends.

Due to the limitations of the free tier of GCP quotas, it progressively over the course of the project whittled down the scale we wanted to operate at due to consistent quota issues pertaining to running out of resources when trying to perform a benchmark. This made apparent we needed to diverge from the specification, and would not be able to use the TPC tools, but would rather have to use components of it in our own inspired process. It became increasingly difficult to make decisions as they were no longer backed by theory or the specification. This caused alot of uncertainty that slowed down progression due to hesitation.

We had issues even running a scale factor of 100, as we encountered problems with quotas that we could not figure out, as it claimed our name nodes were low on resources, despite the console showing we had not reached or exceeded our quotas.

So after a lot of battling with the configuration of Nix and Terraform, and progressively reducing complexity and size of the benchmark queries, Spark nodes and external disk on our GCP environment, given time constraints we defaulted to running a simple proof of concept. Looking back, despite the problems on our 9th semester project with Telenor grossly delaying access to our boxes to perform the performance test, we took the resources and configuration on their system for granted. Taking on Big Data architecture to be scalable and easily buildable alongside the intention of running intricate performance tests was way to ambitious for a masters project, as each are a masters project in their own right. So essentially, despite our limited success with our delivered framework, our not so elegant PyScripts, and our scaled down performance test, we are quite proud at what we achieved, as it has layed a foundation that works and can definitely be improved on as a tool with legitimate uses for the Big Data industry.

8.2 Future Work

Should we continue this project, there are many possibilities to focus on. A glaring issue is to address the resource problems we had on our cluster preventing a larger scale benchmark. We need to investigate our configuration and status of the nodes to determine exactly where the problem lies. It is of course preferable that we attain more resources than just the free tier, as more cores, memory and disk space would certainly enable a larger scale of benchmark. Despite our trouble with running a significantly large benchmark, we did actually manage to answer our first and main problem statement, being that we have created a software framework that makes a reproducible environment, that can run a big data performance test.

We intended to compare the Spark results for Parquet and ORC with results of ORC on Hive and Parquet on Impala. This was to see if the file formats out perform themselves on Spark, when they are used by the tools they were designed to run on. Future work into this idea requires us to finish the configuration of Map Reduce.

Our PySpark code is messy, and not automated in the slightest which isn't user friendly and doesn't abide to the general benchmark guidelines outlined in 3.5. We want to include the scripts into the building of the GCP environment, and use an aliases on the environment to automate the Bash commands and spark-submit commands. We would allow like to redevelop the scripts to use a command line interface to allow easy modification of certain parameters in the scripts making it user friendly and easy to use and read.

Our query selection strategy was mainly focused on finding a diverse selection of queries within the identified reporting queries. Moving forward in conjunction of running a successful performance test, it would be desirable to expand on the query selection to include the different query behaviours, and identify operators that impact performance such as INTERSECT.

One of our problem statements was to run a standardized benchmark, which we don't quite attempt to do given we implement a subset of the TPC-DS benchmark. It would be interesting to perform a valid TPC-DS bench to prove this framework does in fact make performing TPC benchmarks easier with the tool we have created, constructing a sufficient environment to test.

8.3 Project conclusion

Overall, this project has been enjoyable, despite many frustrating goose chases to implement various features. We intended to leave this project with a deep understanding of Parquet and ORC performance, their features and when to select one over another. However we have instead emerged with an understanding of Big Data architecture and configuration, infrastructure as code, benchmarking practices, and the frustrations of resource requirements. This has certainly gotten us much closer to understanding what it means to be a data engineer / dev-ops, though we are not quite satisfied with the outcome and want to pursue it even further.

Bibliography

[1] Michael Armbrust et al. “A View of Cloud Computing”. In: *Commun. ACM* 53.4 (Apr. 2010), pp. 50–58. ISSN: 0001-0782. DOI: 10.1145/1721654.1721672. URL: http://doi.acm.org/10.1145/1721654.1721672.

[2] Tom Breur. “Statistical Power Analysis and the contemporary “crisis” in social sciences”. In: *Journal of Marketing Analytics* 4.2 (2016), pp. 61–65. ISSN: 2050-3326. DOI: 10.1057/s41270-016-0001-3. URL: https://doi.org/10.1057/s41270-016-0001-3.

[3] Y. Brikman. *Terraform: Up and Running : Writing Infrastructure as Code*. O’Reilly Media, 2017. ISBN: 9781491977088. URL: https://books.google.dk/books?id=MLkRMQAACAAJ.

[4] Eelco Dolstra, Merijn de Jonge, and Eelco Visser. “Nix: A Safe and Policy-Free System for Software Deployment”. In: *Proceedings of the 18th USENIX Conference on System Administration*. LISA ’04. Atlanta, GA: USENIX Association, 2004, pp. 79–92. URL: http://dl.acm.org/citation.cfm?id=1052676.1052686.

[5] Eelco Dolstra and Andres Löh. “NixOS: a purely functional Linux distribution”. In: *ICFP*. 2008.

[6] Avrilia Floratou, Umar Farooq Minhas, and Fatma Özcan. “SQL-on-Hadoop: Full Circle Back to Shared-nothing Database Architectures”. In: *Proc. VLDB Endow.* 7.12 (Aug. 2014), pp. 1295–1306. ISSN: 2150-8097. DOI: 10.14778/2732977.2733002. URL: http://dx.doi.org/10.14778/2732977.2733002.

[7] J.U. Gonzalez and S.P.T. Krishnan. *Building Your Next Big Thing with Google Cloud Platform: A Guide for Developers and Enterprise Architects*. Apress, 2015. ISBN: 9781484210048. URL: https://books.google.dk/books?id=BGYnCgAAQBAJ.

[8] Rui Han, Xiaoyi Lu, and Jiangtao Xu. “On big data benchmarking”. In: *Workshop on Big Data Benchmarks, Performance Optimization, and Emerging Hardware*. Springer. 2014, pp. 3–18.

[9] Martin Hilbert. "Big Data for Development: A Review of Promises and Challenges". In: *Development Policy Review* 34.1 (Jan. 2016), pp. 135–174. ISSN: 1467-7679. DOI: 10.1111/dpr.12142. URL: https://doi.org/10.1111/dpr.12142.

[10] Adam Jacobs. "The Pathologies of Big Data". In: *Commun. ACM* 52.8 (Aug. 2009), pp. 36–44. ISSN: 0001-0782. DOI: 10.1145/1536616.1536632. URL: http://doi.acm.org/10.1145/1536616.1536632.

[11] Meikel Poess, Raghunath Othayoth Nambiar, and David Walrath. "Why You Should Run TPC-DS: A Workload Analysis". In: *Proceedings of the 33rd International Conference on Very Large Data Bases*. VLDB '07. Vienna, Austria: VLDB Endowment, 2007, pp. 1138–1149. ISBN: 978-1-59593-649-3. URL: http://dl.acm.org/citation.cfm?id=1325851.1325979.

[12] *Resource quotas*. https://cloud.google.com/compute/quotas. Accessed: 2019-09-12.

[13] Dirk Riehle and Thomas Gross. "Role Model Based Framework Design and Integration". In: *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '98. Vancouver, British Columbia, Canada: ACM, 1998, pp. 117–133. ISBN: 1-58113-005-8. DOI: 10.1145/286936.286951. URL: http://doi.acm.org/10.1145/286936.286951.

[14] Transaction Processing Performance Council (TCP). *TPC BENCHMARK DS - TCP DS Specificaiton V 1.0.0 L*. Last accessed 05 September 2019. 2019. URL: www.tcp.org.

[15] Transaction Processing Performance Council (TCP). *TPC Express Big Bench TPCx BB - TCPx BB V1 3 1.3.0*. Last accessed 05 September 2019. 2019. URL: www.tcp.org.

[16] Ashish Thusoo et al. "Hive: a warehousing solution over a map-reduce framework". In: *Proceedings of the VLDB Endowment* 2.2 (2009), pp. 1626–1629.

[17] tpc. *TPC Benchmarks*. http://www.tpc.org/information/benchmarks.asp. 2019.

[18] Jorge Veiga et al. "Performance evaluation of big data frameworks for large-scale data analytics". In: *2016 IEEE International Conference on Big Data (Big Data)*. IEEE. 2016, pp. 424–431.

[19] Heyong Wang, Wu He, and Feng-Kwei Wang. "Enterprise cloud service architectures". In: *Information Technology and Management* 13.4 (2012), pp. 445–454. ISSN: 1573-7667. DOI: 10.1007/s10799-012-0139-4. URL: https://doi.org/10.1007/s10799-012-0139-4.

[20] Tom White. *Hadoop: The Definitive Guide*. 4th. O'Reilly Media, Inc., 2015. ISBN: 1491901632, 9781491901632.

[21] Matei Zaharia et al. “Spark: Cluster computing with working sets.” In: *HotCloud* 10.10-10 (2010), p. 95.

Appendix A

transform.py

```python
from pyspark.context import SparkContext
from pyspark.sql.session import SparkSession
from pyspark.sql import SQLContext
from pyspark.sql.functions import col
sc = SparkContext('local')
spark = SparkSession(sc)


print("writing parquet files")
df = spark.read.format("csv").load("/data/catalog_sales.csv")
df.write.parquet("/data/parquet_catalog_sales")

df = spark.read.format("csv").load("/data/customer.csv")
df.write.parquet("/data/parquet_customer")

df = spark.read.format("csv").load("/data/customer_address.csv")
df.write.parquet("/data/parquet_customer_address")

df = spark.read.format("csv").load("/data/date_dim.csv")
df.write.parquet("/data/parquet_date_dim")

df = spark.read.format("csv").load("/data/store_sales.csv")
df.write.parquet("/data/parquet_store_sales")

df = spark.read.format("csv").load("/data/web_sales.csv")
df.write.parquet("/data/parquet_web_sales")

df = spark.read.format("csv").load("/data/item.csv")
```

```
df.write.parquet("/data/parquet_item")

df = spark.read.format("csv").load("/data/web_returns.csv")
df.write.parquet("/data/parquet_web_returns")

df = spark.read.format("csv").load("/data/catalog_returns.csv")
df.write.parquet("/data/parquet_catalog_returns")

df = spark.read.format("csv").load("/data/store_returns.csv")
df.write.parquet("/data/parquet_store_returns")
print("parquet files wrote successfully")

print("writing orc files")
df = spark.read.format("csv").load("/data/catalog_sales.csv")
df.write.orc("/data/orc_catalog_sales")

df = spark.read.format("csv").load("/data/customer.csv")
df.write.orc("/data/orc_customer")

df = spark.read.format("csv").load("/data/customer_address.csv")
df.write.orc("/data/orc_customer_address")

df = spark.read.format("csv").load("/data/date_dim.csv")
df.write.orc("/data/orc_date_dim")

df = spark.read.format("csv").load("/data/store_sales.csv")
df.write.orc("/data/orc_store_sales")

df = spark.read.format("csv").load("/data/web_sales.csv")
df.write.orc("/data/orc_web_sales")

df = spark.read.format("csv").load("/data/item.csv")
df.write.orc("/data/orc_item")

df = spark.read.format("csv").load("/data/web_returns.csv")
df.write.orc("/data/orc_web_returns")

df = spark.read.format("csv").load("/data/catalog_returns.csv")
df.write.orc("/data/orc_catalog_returns")

df = spark.read.format("csv").load("/data/store_returns.csv")
```

```
df.write.orc("/data/orc_store_returns")

print("orc files wrote successfully")
```

Appendix B

benchmark.py

```python
import os, sys, time
#from pyspark import SparkConf
from pyspark.context import SparkContext
from pyspark.sql.session import SparkSession
from pyspark.sql import SQLContext
from pyspark.sql.functions import col
sc = SparkContext('local')
spark = SparkSession(sc)
sqlContext = SQLContext(sc)


def set_tables():
    """
    print("registering parquet tables")
    df = spark.read.parquet("hdfs://34.76.28.34:9000/parquet_catalog_sales")
    #df.show(1)
    df = df.select( col("_c0").alias("cs_sold_date_sk"),
    col("_c1").alias("cs_sold_time_sk"), col("_c2").alias("cs_ship_date_sk"), \
    col("_c3").alias("cs_bill_customer_sk"), col("_c4").alias("cs_bill_cdemo_sk"), \
    col("_c5").alias("cs_bill_hdemo_sk"), col("_c6").alias("cs_bill_addr_sk"), \
    col("_c7").alias("cs_ship_customer_sk"), col("_c8").alias("cs_ship_cdemo_sk"), \
    col("_c9").alias("cs_ship_hdemo_sk"), col("_c10").alias("cs_ship_addr_sk"), \
    col("_c11").alias("cs_call_center_sk"), col("_c12").alias("cs_catalog_page_sk"), \
    col("_c13").alias("cs_ship_mode_sk"), col("_c14").alias("cs_warehouse_sk"), \
    col("_c15").alias("cs_item_sk"), col("_c16").alias("cs_promo_sk"), \
    col("_c17").alias("cs_order_number"), col("_c18").alias("cs_quantity"), \
    col("_c19").alias("cs_wholesale_cost"), col("_c20").alias("cs_list_price"), \
    col("_c21").alias("cs_sales_price"), col("_c22").alias("cs_ext_discount_amt"), \
    col("_c23").alias("cs_ext_sales_price"), \
    col("_c24").alias("cs_ext_wholesale_cost"), \
    col("_c25").alias("cs_ext_list_price"), col("_c26").alias("cs_ext_tax"), \
    col("_c27").alias("cs_coupon_amt"), col("_c28").alias("cs_ext_ship_cost"), \
    col("_c29").alias("cs_net_paid"), col("_c30").alias("cs_net_paid_inc_tax"), \
    col("_c31").alias("cs_net_paid_inc_ship"), \
    col("_c32").alias("cs_net_paid_inc_ship_tax"), \
    col("_c33").alias("cs_net_profit"))
    df.registerTempTable("catalog_sales")

    df = spark.read.parquet("hdfs://34.76.28.34:9000/parquet_customer")
    df = df.select( col("_c0").alias("c_customer_sk"), \
    col("_c1").alias("c_customer_id (B)"), \
    col("_c2").alias("c_current_cdemo_sk"), \
    col("_c3").alias("c_current_hdemo_sk"), \
    col("_c4").alias("c_current_addr_sk"), \
    col("_c5").alias("c_first_shipto_date_sk"), \
    col("_c6").alias("c_first_sales_date_sk"), \
    col("_c7").alias("c_salutation"), col("_c8").alias("c_first_name"), \
    col("_c9").alias("c_last_name"), \
    col("_c10").alias("c_preferred_cust_flag"), \
    col("_c11").alias("c_birth_day"), col("_c12").alias("c_birth_month"), \
    col("_c13").alias("c_birth_year"), \
    col("_c14").alias("c_birth_country"), col("_c15").alias("c_login"), \
    col("_c16").alias("c_email_address"), \
    col("_c17").alias("c_last_review_date_sk"))
    df.registerTempTable("customer")

    df = spark.read.parquet("hdfs://34.76.28.34:9000/parquet_customer_address")
    df = df.select( col("_c0").alias("ca_address_sk"), \
    col("_c1").alias("ca_address_id (B"), \
    col("_c2").alias("ca_street_number"), \
    col("_c3").alias("ca_street_name"), \
    col("_c4").alias("ca_street_type"), \
    col("_c5").alias("ca_suite_number"), \
    col("_c6").alias("ca_city"), col("_c7").alias("ca_county"), \
    col("_c8").alias("ca_state"), \
    col("_c9").alias("ca_zip"), col("_c10").alias("ca_country"), \
    col("_c11").alias("ca_gmt_offset"), \
    col("_c12").alias("ca_location_type"))
    df.registerTempTable("customer_address")
```

```
        df = spark.read.parquet("hdfs://34.76.28.34:9000/parquet_date_dim")
        df = df.select( col("_c0").alias("d_date_sk"), \
        col("_c1").alias("d_date_id (B)"), col("_c2").alias("d_date"), \
        col("_c3").alias("d_month_seq"), col("_c4").alias("d_week_seq"), \
        col("_c5").alias("d_quarter_seq"), col("_c6").alias("d_year"), \
        col("_c7").alias("d_dow"), col("_c8").alias("d_moy"), \
        col("_c9").alias("d_dom"), col("_c10").alias("d_qoy"), \
        col("_c11").alias("d_fy_year"), col("_c12").alias("d_fy_quarter_seq"), \
        col("_c13").alias("d_fy_week_seq"), col("_c14").alias("d_day_name"), \
        col("_c15").alias("d_quarter_name"), col("_c16").alias("d_holiday"), \
        col("_c17").alias("d_weekend"), col("_c18").alias("d_following_holiday"), \
        col("_c19").alias("d_first_dom"), col("_c20").alias("d_last_dom"), \
        col("_c21").alias("d_same_day_ly"), col("_c22").alias("d_same_day_lq"), \
        col("_c23").alias("d1nt_day"), col("_c24").alias("d_current_week"), \
        col("_c25").alias("d_current_month"), col("_c26").alias("d_current_quarter"), \
        col("_c27").alias("d_current_year"))
        df.registerTempTable("date_dim")

        df = spark.read.parquet("hdfs://34.76.28.34:9000/parquet_store_sales")
        df = df.select( col("_c0").alias("ss_sold_date_sk"), \
        col("_c1").alias("ss_sold_time_sk"), col("_c2").alias("ss_item_sk"), \
        col("_c3").alias("ss_customer_sk"), col("_c4").alias("ss_cdemo_sk"), \
        col("_c5").alias("ss_hdemo_sk"), col("_c6").alias("ss_addr_sk"), \
        col("_c7").alias("ss_store_sk"), col("_c8").alias("ss_promo_sk"), \
        col("_c9").alias("ss_ticket_number"), col("_c10").alias("ss_quantity"), \
        col("_c11").alias("ss_wholesale_cost"), col("_c12").alias("ss_list_price"), \
        col("_c13").alias("ss_sales_price"), \
        col("_c14").alias("ss_ext_discount_amt"), \
        col("_c15").alias("ss_ext_sales_price"), \
        col("_c16").alias("ss_ext_wholesale_cost"), \
        col("_c17").alias("ss_ext_list_price"), col("_c18").alias("ss_ext_tax"), \
        col("_c19").alias("ss_coupon_amt"), col("_c20").alias("ss_net_paid"), \
        col("_c21").alias("ss_net_paid_inc_tax"), col("_c22").alias("ss_net_profit"))
        df.registerTempTable("store_sales")

        df = spark.read.parquet("hdfs://34.76.28.34:9000/parquet_web_sales")
        df = df.select( col("_c0").alias("ws_sold_date_sk"), \
        col("_c1").alias("ws_sold_time_sk"), col("_c2").alias("ws_ship_date_sk"), \
        col("_c3").alias("ws_item_sk"), col("_c4").alias("ws_bill_customer_sk"), \
        col("_c5").alias("ws_bill_cdemo_sk"), col("_c6").alias("ws_bill_hdemo_sk"), \
        col("_c7").alias("ws_bill_addr_sk"), \
        col("_c8").alias("ws_ship_customer_sk"), \
        col("_c9").alias("ws_ship_cdemo_sk"), \
        col("_c10").alias("ws_ship_hdemo_sk"), \
        col("_c11").alias("ws_ship_addr_sk"), \
        col("_c12").alias("ws_web_page_sk"), \
        col("_c13").alias("ws_web_site_sk"), \
        col("_c14").alias("ws_ship_mode_sk"), \
        col("_c15").alias("ws_warehouse_sk"), \
        col("_c16").alias("ws_promo_sk"), \
        col("_c17").alias("ws_order_number"), \
        col("_c18").alias("ws_quantity"), \
        col("_c19").alias("ws_wholesale_cost"), \
        col("_c20").alias("ws_list_price"), \
        col("_c21").alias("ws_sales_price"), \
        col("_c22").alias("ws_ext_discount_amt"), \
        col("_c23").alias("ws_ext_sales_price"), \
        col("_c24").alias("ws_ext_wholesale_cost"), \
        col("_c25").alias("ws_ext_list_price"), \
        col("_c26").alias("ws_ext_tax"), \
        col("_c27").alias("ws_coupon_amt"), \
        col("_c28").alias("ws_ext_ship_cost"), \
        col("_c29").alias("ws_net_paid"), \
        col("_c30").alias("ws_net_paid_inc_tax"), \
        col("_c31").alias("ws_net_paid_inc_ship"), \
        col("_c32").alias("ws_net_paid_inc_ship_tax"), \
        col("_c33").alias("ws_net_profit"))
        df.registerTempTable("web_sales")

        df = spark.read.parquet("hdfs://34.76.28.34:9000/parquet_item")
        df = df.select( col("_c0").alias("i_item_sk"), \
        col("_c1").alias("i_item_id"), col("_c2").alias("i_rec_start_date"), \
        col("_c3").alias("i_rec_end_date"), col("_c4").alias("i_item_desc"), \
        col("_c5").alias("i_current_price"), col("_c6").alias("i_wholesale_cost"), \
        col("_c7").alias("i_brand_id"), col("_c8").alias("i_brand"), \
        col("_c9").alias("i_class_id"), col("_c10").alias("i_class"), \
        col("_c11").alias("i_category_id"), col("_c12").alias("i_category"), \
        col("_c13").alias("i_manufact_id"), col("_c14").alias("i_manufact"), \
        col("_c15").alias("i_size"), col("_c16").alias("i_formulation"), \
        col("_c17").alias("i_color"), col("_c18").alias("i_units"), \
        col("_c19").alias("i_container"), col("_c20").alias("i_manager_id"), \
        col("_c21").alias("i_product_name"))
        df.registerTempTable("item")

        df = spark.read.parquet("hdfs://34.76.28.34:9000/parquet_web_returns")
        df = df.select( col("_c0").alias("wr_returned_date_sk"), \
        col("_c1").alias("wr_returned_time_sk"), col("_c2").alias("wr_item_sk"), \
        col("_c3").alias("wr_refunded_customer_sk"), \
        col("_c4").alias("wr_refunded_cdemo_sk"), \
        col("_c5").alias("wr_refunded_hdemo_sk"), \
        col("_c6").alias("wr_refunded_addr_sk"), \
        col("_c7").alias("wr_returning_customer_sk"), \
        col("_c8").alias("wr_returning_cdemo_sk"), \
        col("_c9").alias("wr_returning_hdemo_sk"), \
        col("_c10").alias("wr_returning_addr_sk"), \
        col("_c11").alias("wr_web_page_sk"), \
        col("_c12").alias("wr_reason_sk"), \
        col("_c13").alias("wr_order_number"), \
        col("_c14").alias("wr_return_quantity"), \
        col("_c15").alias("wr_return_amt"), \
        col("_c16").alias("wr_return_tax"), \
        col("_c17").alias("wr_return_amt_inc_tax"), \
        col("_c18").alias("wr_fee"), col("_c19").alias("wr_return_ship_cost"), \
        col("_c20").alias("wr_refunded_cash"), \
        col("_c21").alias("wr_reversed_charge"), \
        col("_c22").alias("wr_account_credit"), \
        col("_c23").alias("wr_net_loss"))
        df.registerTempTable("web_returns")

        df = spark.read.parquet("hdfs://34.76.28.34:9000/parquet_catalog_returns")
        df = df.select( col("_c0").alias("cr_returned_date_sk"), \
        col("_c1").alias("cr_returned_time_sk"), col("_c2").alias("cr_item_sk"), \
        col("_c3").alias("cr_refunded_customer_sk"), \
        col("_c4").alias("cr_refunded_cdemo_sk"), \
        col("_c5").alias("cr_refunded_hdemo_sk"), \
        col("_c6").alias("cr_refunded_addr_sk"), \
        col("_c7").alias("cr_returning_customer_sk"), \
        col("_c8").alias("cr_returning_cdemo_sk"), \
        col("_c9").alias("cr_returning_hdemo_sk"), \
        col("_c10").alias("cr_returning_addr_sk"), \
        col("_c11").alias("cr_call_center_sk"), \
        col("_c12").alias("cr_catalog_page_sk"), \
        col("_c13").alias("cr_ship_mode_sk"), \
        col("_c14").alias("cr_warehouse_sk"), \
        col("_c15").alias("cr_reason_sk"), col("_c16").alias("cr_order_number"), \
        col("_c17").alias("cr_return_quantity"), \
        col("_c18").alias("cr_return_amount"), \
        col("_c19").alias("cr_return_tax"), \
        col("_c20").alias("cr_return_amt_inc_tax"), \
        col("_c21").alias("cr_fee"), col("_c22").alias("cr_return_ship_cost"), \
        col("_c23").alias("cr_refunded_cash"), \
        col("_c24").alias("cr_reversed_charge"), \
        col("_c25").alias("cr_store_credit"), \
        col("_c26").alias("cr_net_loss"))
        df.registerTempTable("catalog_returns")

        df = spark.read.parquet("hdfs://34.76.28.34:9000/parquet_store_returns")
        df = df.select( col("_c0").alias("sr_returned_date_sk"), \
        col("_c1").alias("sr_return_time_sk"), col("_c2").alias("sr_item_sk"), \
        col("_c3").alias("sr_customer_sk"), col("_c4").alias("sr_cdemo_sk"), \
        col("_c5").alias("sr_hdemo_sk"), col("_c6").alias("sr_addr_sk"), \
        col("_c7").alias("sr_store_sk"), col("_c8").alias("sr_reason_sk"), \
        col("_c9").alias("sr_ticket_number"), \
        col("_c10").alias("sr_return_quantity"), \
        col("_c11").alias("sr_return_amt"), \
        col("_c12").alias("sr_return_tax"), \
        col("_c13").alias("sr_return_amt_inc_tax"), \
        col("_c14").alias("sr_fee"), col("_c15").alias("sr_return_ship_cost"), \
        col("_c16").alias("sr_refunded_cash"), \
        col("_c17").alias("sr_reversed_charge"), \
        col("_c18").alias("sr_store_credit"), \
        col("_c19").alias("sr_net_loss"))
        df.registerTempTable("store_returns")
        print("parquet tables registered")
        """
        print("registering orc tables")
        df = spark.read.orc("hdfs://34.76.28.34:9000/orc_catalog_sales")
        #df.show(1)
        df = df.select( col("_c0").alias("cs_sold_date_sk"), \
        col("_c1").alias("cs_sold_time_sk"), col("_c2").alias("cs_ship_date_sk"), \
        col("_c3").alias("cs_bill_customer_sk"), \
        col("_c4").alias("cs_bill_cdemo_sk"), \
        col("_c5").alias("cs_bill_hdemo_sk"), col("_c6").alias("cs_bill_addr_sk"), \
        col("_c7").alias("cs_ship_customer_sk"), \
        col("_c8").alias("cs_ship_cdemo_sk"), col("_c9").alias("cs_ship_hdemo_sk"), \
        col("_c10").alias("cs_ship_addr_sk"), \
        col("_c11").alias("cs_call_center_sk"), \
        col("_c12").alias("cs_catalog_page_sk"), \
        col("_c13").alias("cs_ship_mode_sk"), \
        col("_c14").alias("cs_warehouse_sk"), col("_c15").alias("cs_item_sk"), \
        col("_c16").alias("cs_promo_sk"), col("_c17").alias("cs_order_number"), \
        col("_c18").alias("cs_quantity"), col("_c19").alias("cs_wholesale_cost"), \
        col("_c20").alias("cs_list_price"), col("_c21").alias("cs_sales_price"), \
        col("_c22").alias("cs_ext_discount_amt"), \
        col("_c23").alias("cs_ext_sales_price"), \
        col("_c24").alias("cs_ext_wholesale_cost"), \
        col("_c25").alias("cs_ext_list_price"), col("_c26").alias("cs_ext_tax"), \
        col("_c27").alias("cs_coupon_amt"), col("_c28").alias("cs_ext_ship_cost"), \
        col("_c29").alias("cs_net_paid"), col("_c30").alias("cs_net_paid_inc_tax"), \
        col("_c31").alias("cs_net_paid_inc_ship"), \
        col("_c32").alias("cs_net_paid_inc_ship_tax"), \
        col("_c33").alias("cs_net_profit"))
        df.registerTempTable("catalog_sales")

        df = spark.read.orc("hdfs://34.76.28.34:9000/orc_customer")
        df = df.select( col("_c0").alias("c_customer_sk"), \
        col("_c1").alias("c_customer_id (B)"), \
        col("_c2").alias("c_current_cdemo_sk"), \
        col("_c3").alias("c_current_hdemo_sk"), \
        col("_c4").alias("c_current_addr_sk"), \
        col("_c5").alias("c_first_shipto_date_sk"), \
        col("_c6").alias("c_first_sales_date_sk"), \
        col("_c7").alias("c_salutation"), \
        col("_c8").alias("c_first_name"), \
        col("_c9").alias("c_last_name"), \
        col("_c10").alias("c_preferred_cust_flag"), \
        col("_c11").alias("c_birth_day"), \
        col("_c12").alias("c_birth_month"), \
        col("_c13").alias("c_birth_year"), \
        col("_c14").alias("c_birth_country"), \
        col("_c15").alias("c_login"), \
        col("_c16").alias("c_email_address"), \
        col("_c17").alias("c_last_review_date_sk"))
        df.registerTempTable("customer")
```

```

    df = spark.read.orc("hdfs://34.76.28.34:9000/orc_customer_address")
    df = df.select( col("_c0").alias("ca_address_sk"), \
    col("_c1").alias("ca_address_id (B"), \
    col("_c2").alias("ca_street_number"), \
    col("_c3").alias("ca_street_name"), \
    col("_c4").alias("ca_street_type"), \
    col("_c5").alias("ca_suite_number"), \
    col("_c6").alias("ca_city"), col("_c7").alias("ca_county"), \
    col("_c8").alias("ca_state"), col("_c9").alias("ca_zip"), \
    col("_c10").alias("ca_country"), col("_c11").alias("ca_gmt_offset"), \
    col("_c12").alias("ca_location_type"))
    df.registerTempTable("customer_address")

    df = spark.read.orc("hdfs://34.76.28.34:9000/orc_date_dim")
    df = df.select( col("_c0").alias("d_date_sk"), \
    col("_c1").alias("d_date_id (B)"), col("_c2").alias("d_date"), \
    col("_c3").alias("d_month_seq"), col("_c4").alias("d_week_seq"), \
    col("_c5").alias("d_quarter_seq"), col("_c6").alias("d_year"), \
    col("_c7").alias("d_dow"), col("_c8").alias("d_moy"), \
    col("_c9").alias("d_dom"), col("_c10").alias("d_qoy"), \
    col("_c11").alias("d_fy_year"), col("_c12").alias("d_fy_quarter_seq"), \
    col("_c13").alias("d_fy_week_seq"), col("_c14").alias("d_day_name"), \
    col("_c15").alias("d_quarter_name"), col("_c16").alias("d_holiday"), \
    col("_c17").alias("d_weekend"), col("_c18").alias("d_following_holiday"), \
    col("_c19").alias("d_first_dom"), col("_c20").alias("d_last_dom"), \
    col("_c21").alias("d_same_day_ly"), col("_c22").alias("d_same_day_lq"),\
    col("_c23").alias("d1nt_day"), col("_c24").alias("d_current_week"), \
    col("_c25").alias("d_current_month"), \
    col("_c26").alias("d_current_quarter"), col("_c27").alias("d_current_year"))
    df.registerTempTable("date_dim")

    df = spark.read.orc("hdfs://34.76.28.34:9000/orc_store_sales")
    df = df.select( col("_c0").alias("ss_sold_date_sk"), \
    col("_c1").alias("ss_sold_time_sk"), col("_c2").alias("ss_item_sk"), \
    col("_c3").alias("ss_customer_sk"), col("_c4").alias("ss_cdemo_sk"), \
    col("_c5").alias("ss_hdemo_sk"), col("_c6").alias("ss_addr_sk"), \
    col("_c7").alias("ss_store_sk"), col("_c8").alias("ss_promo_sk"), \
    col("_c9").alias("ss_ticket_number"), col("_c10").alias("ss_quantity"), \
    col("_c11").alias("ss_wholesale_cost"), \
    col("_c12").alias("ss_list_price"), \
    col("_c13").alias("ss_sales_price"), \
    col("_c14").alias("ss_ext_discount_amt"), \
    col("_c15").alias("ss_ext_sales_price"), \
    col("_c16").alias("ss_ext_wholesale_cost"), \
    col("_c17").alias("ss_ext_list_price"), \
    col("_c18").alias("ss_ext_tax"), \
    col("_c19").alias("ss_coupon_amt"), \
    col("_c20").alias("ss_net_paid"), \
    col("_c21").alias("ss_net_paid_inc_tax"), \
    col("_c22").alias("ss_net_profit"))
    df.registerTempTable("store_sales")

    df = spark.read.orc("hdfs://34.76.28.34:9000/orc_web_sales")
    df = df.select( col("_c0").alias("ws_sold_date_sk"), \
    col("_c1").alias("ws_sold_time_sk"), col("_c2").alias("ws_ship_date_sk"), \
    col("_c3").alias("ws_item_sk"), col("_c4").alias("ws_bill_customer_sk"), \
    col("_c5").alias("ws_bill_cdemo_sk"), col("_c6").alias("ws_bill_hdemo_sk"), \
    col("_c7").alias("ws_bill_addr_sk"), col("_c8").alias("ws_ship_customer_sk"), \
    col("_c9").alias("ws_ship_cdemo_sk"), col("_c10").alias("ws_ship_hdemo_sk"), \
    col("_c11").alias("ws_ship_addr_sk"), col("_c12").alias("ws_web_page_sk"), \
    col("_c13").alias("ws_web_site_sk"), col("_c14").alias("ws_ship_mode_sk"), \
    col("_c15").alias("ws_warehouse_sk"), col("_c16").alias("ws_promo_sk"), \
    col("_c17").alias("ws_order_number"), col("_c18").alias("ws_quantity"), \
    col("_c19").alias("ws_wholesale_cost"), col("_c20").alias("ws_list_price"), \
    col("_c21").alias("ws_sales_price"), col("_c22").alias("ws_ext_discount_amt"), \
    col("_c23").alias("ws_ext_sales_price"), \
    col("_c24").alias("ws_ext_wholesale_cost"), \
    col("_c25").alias("ws_ext_list_price"), col("_c26").alias("ws_ext_tax"), \
    col("_c27").alias("ws_coupon_amt"), col("_c28").alias("ws_ext_ship_cost"), \
    col("_c29").alias("ws_net_paid"), col("_c30").alias("ws_net_paid_inc_tax"), \
    col("_c31").alias("ws_net_paid_inc_ship"), \
    col("_c32").alias("ws_net_paid_inc_ship_tax"), col("_c33").alias("ws_net_profit"))
    df.registerTempTable("web_sales")

    df = spark.read.orc("hdfs://34.76.28.34:9000/orc_item")
    df = df.select( col("_c0").alias("i_item_sk"), col("_c1").alias("i_item_id"), \
    col("_c2").alias("i_rec_start_date"), col("_c3").alias("i_rec_end_date"), \
    col("_c4").alias("i_item_desc"), col("_c5").alias("i_current_price"), \
    col("_c6").alias("i_wholesale_cost"), col("_c7").alias("i_brand_id"), \
    col("_c8").alias("i_brand"), col("_c9").alias("i_class_id"), \
    col("_c10").alias("i_class"), col("_c11").alias("i_category_id"), \
    col("_c12").alias("i_category"), col("_c13").alias("i_manufact_id"), \
    col("_c14").alias("i_manufact"), col("_c15").alias("i_size"), \
    col("_c16").alias("i_formulation"), col("_c17").alias("i_color"), \
    col("_c18").alias("i_units"), col("_c19").alias("i_container"), \
    col("_c20").alias("i_manager_id"), col("_c21").alias("i_product_name"))
    df.registerTempTable("item")

    df = spark.read.orc("hdfs://34.76.28.34:9000/orc_web_returns")
    df = df.select( col("_c0").alias("wr_returned_date_sk"), \
    col("_c1").alias("wr_returned_time_sk"), \
    col("_c2").alias("wr_item_sk"), \
    col("_c3").alias("wr_refunded_customer_sk"), \
    col("_c4").alias("wr_refunded_cdemo_sk"), \
    col("_c5").alias("wr_refunded_hdemo_sk"), \
    col("_c6").alias("wr_refunded_addr_sk"), \
    col("_c7").alias("wr_returning_customer_sk"), \
    col("_c8").alias("wr_returning_cdemo_sk"), \
    col("_c9").alias("wr_returning_hdemo_sk"), \
    col("_c10").alias("wr_returning_addr_sk"), \
    col("_c11").alias("wr_web_page_sk"), \
    col("_c12").alias("wr_reason_sk"), \
    col("_c13").alias("wr_order_number"), \
    col("_c14").alias("wr_return_quantity"), \
    col("_c15").alias("wr_return_amt"), \
    col("_c16").alias("wr_return_tax"), \
    col("_c17").alias("wr_return_amt_inc_tax"), \
    col("_c18").alias("wr_fee"), \
    col("_c19").alias("wr_return_ship_cost"), \
    col("_c20").alias("wr_refunded_cash"), \
    col("_c21").alias("wr_reversed_charge"), \
    col("_c22").alias("wr_account_credit"), col("_c23").alias("wr_net_loss"))
    df.registerTempTable("web_returns")

    df = spark.read.orc("hdfs://34.76.28.34:9000/orc_catalog_returns")
    df = df.select( col("_c0").alias("cr_returned_date_sk"), \
    col("_c1").alias("cr_returned_time_sk"), \
    col("_c2").alias("cr_item_sk"), \
    col("_c3").alias("cr_refunded_customer_sk"), \
    col("_c4").alias("cr_refunded_cdemo_sk"), \
    col("_c5").alias("cr_refunded_hdemo_sk"), \
    col("_c6").alias("cr_refunded_addr_sk"), \
    col("_c7").alias("cr_returning_customer_sk"), \
    col("_c8").alias("cr_returning_cdemo_sk"), \
    col("_c9").alias("cr_returning_hdemo_sk"), \
    col("_c10").alias("cr_returning_addr_sk"), \
    col("_c11").alias("cr_call_center_sk"), \
    col("_c12").alias("cr_catalog_page_sk"), \
    col("_c13").alias("cr_ship_mode_sk"), \
    col("_c14").alias("cr_warehouse_sk"), \
    col("_c15").alias("cr_reason_sk"), \
    col("_c16").alias("cr_order_number"), \
    col("_c17").alias("cr_return_quantity"), \
    col("_c18").alias("cr_return_amount"), \
    col("_c19").alias("cr_return_tax"), \
    col("_c20").alias("cr_return_amt_inc_tax"), \
    col("_c21").alias("cr_fee"), \
    col("_c22").alias("cr_return_ship_cost"), \
    col("_c23").alias("cr_refunded_cash"), \
    col("_c24").alias("cr_reversed_charge"), \
    col("_c25").alias("cr_store_credit"), col("_c26").alias("cr_net_loss"))
    df.registerTempTable("catalog_returns")

    df = spark.read.orc("hdfs://34.76.28.34:9000/orc_store_returns")
    df = df.select( col("_c0").alias("sr_returned_date_sk"), \
    col("_c1").alias("sr_return_time_sk"), col("_c2").alias("sr_item_sk"), \
    col("_c3").alias("sr_customer_sk"), col("_c4").alias("sr_cdemo_sk"), \
    col("_c5").alias("sr_hdemo_sk"), col("_c6").alias("sr_addr_sk"), \
    col("_c7").alias("sr_store_sk"), col("_c8").alias("sr_reason_sk"), \
    col("_c9").alias("sr_ticket_number"), col("_c10").alias("sr_return_quantity"), \
    col("_c11").alias("sr_return_amt"), col("_c12").alias("sr_return_tax"), \
    col("_c13").alias("sr_return_amt_inc_tax"), col("_c14").alias("sr_fee"), \
    col("_c15").alias("sr_return_ship_cost"), col("_c16").alias("sr_refunded_cash"), \
    col("_c17").alias("sr_reversed_charge"), col("_c18").alias("sr_store_credit"), \
    col("_c19").alias("sr_net_loss"))
    df.registerTempTable("store_returns")
    print("orc tables registered")


def execute_queries():
    print("starting query 1")
    query_start_time = time.time()
    query1 = sqlContext.sql("""SELECT
  ca_zip,
  sum(cs_sales_price)
FROM catalog_sales, customer, customer_address, date_dim
WHERE cs_bill_customer_sk = c_customer_sk
  AND c_current_addr_sk = ca_address_sk
  AND (substr(ca_zip, 1, 5) IN ('85669', '86197', '88274', '83405', '86475',
                              '85392', '85460', '80348', '81792')
  OR ca_state IN ('CA', 'WA', 'GA')
  OR cs_sales_price > 500)
  AND cs_sold_date_sk = d_date_sk
  AND d_qoy = 2 AND d_year = 2001
GROUP BY ca_zip
ORDER BY ca_zip
LIMIT 100""")
    query1.show()
    query_end_time = time.time()
    query_elapsed_time1 = (query_end_time - query_start_time)
    print("query elapsed time: " + str(query_elapsed_time1))
    print("\n")

    print("starting query 2")
    query_start_time = time.time()
    query2 = sqlContext.sql("""SELECT count(*)
FROM (
       SELECT DISTINCT
         c_last_name,
         c_first_name,
         d_date
       FROM store_sales, date_dim, customer
       WHERE store_sales.ss_sold_date_sk = date_dim.d_date_sk
         AND store_sales.ss_customer_sk = customer.c_customer_sk
         AND d_month_seq BETWEEN 1200 AND 1200 + 11
       INTERSECT
       SELECT DISTINCT
         c_last_name,
         c_first_name,
         d_date
       FROM catalog_sales, date_dim, customer
       WHERE catalog_sales.cs_sold_date_sk = date_dim.d_date_sk
         AND catalog_sales.cs_bill_customer_sk = customer.c_customer_sk
         AND d_month_seq BETWEEN 1200 AND 1200 + 11
```

```
        INTERSECT
        SELECT DISTINCT
          c_last_name,
          c_first_name,
          d_date
        FROM web_sales, date_dim, customer
        WHERE web_sales.ws_sold_date_sk = date_dim.d_date_sk
          AND web_sales.ws_bill_customer_sk = customer.c_customer_sk
          AND d_month_seq BETWEEN 1200 AND 1200 + 11
      ) hot_cust
LIMIT 100""")
    query2.show()
    query_end_time = time.time()
    query_elapsed_time2 = (query_end_time - query_start_time)
    print("query elapsed time: " + str(query_elapsed_time2))
    print("\n")

    print("starting query 3")
    query_start_time = time.time()
    query3 = sqlContext.sql("""SELECT
  'web' AS channel,
  web.item,
  web.return_ratio,
  web.return_rank,
  web.currency_rank
FROM (
        SELECT
          item,
          return_ratio,
          currency_ratio,
          rank()
          OVER (
            ORDER BY return_ratio) AS return_rank,
          rank()
          OVER (
            ORDER BY currency_ratio) AS currency_rank
        FROM
          (SELECT
            ws.ws_item_sk AS item,
            (cast(sum(coalesce(wr.wr_return_quantity, 0)) AS DECIMAL(15, 4)) /
              cast(sum(coalesce(ws.ws_quantity, 0)) AS DECIMAL(15, 4))) AS return_ratio,
            (cast(sum(coalesce(wr.wr_return_amt, 0)) AS DECIMAL(15, 4)) /
              cast(sum(coalesce(ws.ws_net_paid, 0)) AS DECIMAL(15, 4))) \
              AS currency_ratio
          FROM
            web_sales ws LEFT OUTER JOIN web_returns wr
              ON (ws.ws_order_number = wr.wr_order_number AND
              ws.ws_item_sk = wr.wr_item_sk)
            , date_dim
          WHERE
            wr.wr_return_amt > 10000
              AND ws.ws_net_profit > 1
              AND ws.ws_net_paid > 0
              AND ws.ws_quantity > 0
              AND ws_sold_date_sk = d_date_sk
              AND d_year = 2001
              AND d_moy = 12
          GROUP BY ws.ws_item_sk
          ) in_web
      ) web
WHERE (web.return_rank <= 10 OR web.currency_rank <= 10)
UNION
SELECT
  'catalog' AS channel,
  catalog.item,
  catalog.return_ratio,
  catalog.return_rank,
  catalog.currency_rank
FROM (
        SELECT
          item,
          return_ratio,
          currency_ratio,
          rank()
          OVER (
            ORDER BY return_ratio) AS return_rank,
          rank()
          OVER (
            ORDER BY currency_ratio) AS currency_rank
        FROM
          (SELECT
            cs.cs_item_sk AS item,
            (cast(sum(coalesce(cr.cr_return_quantity, 0)) AS DECIMAL(15, 4)) /
              cast(sum(coalesce(cs.cs_quantity, 0)) AS DECIMAL(15, 4))) \
              AS return_ratio,
            (cast(sum(coalesce(cr.cr_return_amount, 0)) AS DECIMAL(15, 4)) /
              cast(sum(coalesce(cs.cs_net_paid, 0)) AS DECIMAL(15, 4))) \
              AS currency_ratio
          FROM
            catalog_sales cs LEFT OUTER JOIN catalog_returns cr
              ON (cs.cs_order_number = cr.cr_order_number AND
              cs.cs_item_sk = cr.cr_item_sk)
            , date_dim
          WHERE
            cr.cr_return_amount > 10000
              AND cs.cs_net_profit > 1
              AND cs.cs_net_paid > 0
              AND cs.cs_quantity > 0
              AND cs_sold_date_sk = d_date_sk
              AND d_year = 2001
              AND d_moy = 12
          GROUP BY cs.cs_item_sk
          ) in_cat
      ) catalog
WHERE (catalog.return_rank <= 10 OR catalog.currency_rank <= 10)
UNION
SELECT
  'store' AS channel,
  store.item,
  store.return_ratio,
  store.return_rank,
  store.currency_rank
FROM (
       SELECT
         item,
         return_ratio,
         currency_ratio,
         rank()
         OVER (
           ORDER BY return_ratio) AS return_rank,
         rank()
         OVER (
           ORDER BY currency_ratio) AS currency_rank
       FROM
         (SELECT
           sts.ss_item_sk AS item,
            (cast(sum(coalesce(sr.sr_return_quantity, 0)) AS DECIMAL(15, 4)) /
              cast(sum(coalesce(sts.ss_quantity, 0)) AS DECIMAL(15, 4))) \
              AS return_ratio,
            (cast(sum(coalesce(sr.sr_return_amt, 0)) AS DECIMAL(15, 4)) /
              cast(sum(coalesce(sts.ss_net_paid, 0)) AS DECIMAL(15, 4))) \
              AS currency_ratio
         FROM
            store_sales sts LEFT OUTER JOIN store_returns sr
              ON (sts.ss_ticket_number = sr.sr_ticket_number AND \
              sts.ss_item_sk = sr.sr_item_sk)
            , date_dim
         WHERE
            sr.sr_return_amt > 10000
              AND sts.ss_net_profit > 1
              AND sts.ss_net_paid > 0
              AND sts.ss_quantity > 0
              AND ss_sold_date_sk = d_date_sk
              AND d_year = 2001
              AND d_moy = 12
         GROUP BY sts.ss_item_sk
         ) in_store
      ) store
WHERE (store.return_rank <= 10 OR store.currency_rank <= 10)
ORDER BY 1, 4, 5
LIMIT 100""")
    query3.show()
    query_end_time = time.time()
    query_elapsed_time3 = (query_end_time - query_start_time)
    print("query elapsed time: " + str(query_elapsed_time3))
    print("\n")

    print("starting query 4")
    query_start_time = time.time()
    query4 = sqlContext.sql("""WITH ss AS (
  SELECT
    i_item_id,
    sum(ss_ext_sales_price) total_sales
  FROM
```

```
    store_sales, date_dim, customer_address, item
  WHERE
    i_item_id IN (SELECT i_item_id
    FROM item
    WHERE i_color IN ('slate', 'blanched', 'burnished'))
      AND ss_item_sk = i_item_sk
      AND ss_sold_date_sk = d_date_sk
      AND d_year = 2001
      AND d_moy = 2
      AND ss_addr_sk = ca_address_sk
      AND ca_gmt_offset = -5
  GROUP BY i_item_id),
    cs AS (
    SELECT
      i_item_id,
      sum(cs_ext_sales_price) total_sales
    FROM
      catalog_sales, date_dim, customer_address, item
    WHERE
      i_item_id IN (SELECT i_item_id
      FROM item
      WHERE i_color IN ('slate', 'blanched', 'burnished'))
        AND cs_item_sk = i_item_sk
        AND cs_sold_date_sk = d_date_sk
        AND d_year = 2001
        AND d_moy = 2
        AND cs_bill_addr_sk = ca_address_sk
        AND ca_gmt_offset = -5
    GROUP BY i_item_id),
    ws AS (
    SELECT
      i_item_id,
      sum(ws_ext_sales_price) total_sales
    FROM
      web_sales, date_dim, customer_address, item
    WHERE
      i_item_id IN (SELECT i_item_id
      FROM item
      WHERE i_color IN ('slate', 'blanched', 'burnished'))
        AND ws_item_sk = i_item_sk
        AND ws_sold_date_sk = d_date_sk
        AND d_year = 2001
        AND d_moy = 2
        AND ws_bill_addr_sk = ca_address_sk
        AND ca_gmt_offset = -5
    GROUP BY i_item_id)
SELECT
  i_item_id,
  sum(total_sales) total_sales
FROM (SELECT *
      FROM ss
      UNION ALL
      SELECT *
      FROM cs
      UNION ALL
      SELECT *
      FROM ws) tmp1
GROUP BY i_item_id
ORDER BY total_sales
LIMIT 100""")
    query4.show()
    query_end_time = time.time()
    query_elapsed_time4 = (query_end_time - query_start_time)
    print("query elapsed time: " + str(query_elapsed_time4))
    print("\n")

    print("starting query 5")
    query_start_time = time.time()
    query5 = sqlContext.sql("""WITH ss AS (
  SELECT
    i_item_id,
    sum(ss_ext_sales_price) total_sales
  FROM store_sales, date_dim, customer_address, item
  WHERE
    i_item_id IN (SELECT i_item_id
    FROM item
    WHERE i_category IN ('Music'))
      AND ss_item_sk = i_item_sk
      AND ss_sold_date_sk = d_date_sk
      AND d_year = 1998
      AND d_moy = 9
      AND ss_addr_sk = ca_address_sk
      AND ca_gmt_offset = -5
  GROUP BY i_item_id),
    cs AS (
    SELECT
      i_item_id,
      sum(cs_ext_sales_price) total_sales
    FROM catalog_sales, date_dim, customer_address, item
    WHERE
      i_item_id IN (SELECT i_item_id
      FROM item
      WHERE i_category IN ('Music'))
        AND cs_item_sk = i_item_sk
        AND cs_sold_date_sk = d_date_sk
        AND d_year = 1998
        AND d_moy = 9
        AND cs_bill_addr_sk = ca_address_sk
        AND ca_gmt_offset = -5
    GROUP BY i_item_id),
    ws AS (
    SELECT
      i_item_id,
      sum(ws_ext_sales_price) total_sales
    FROM web_sales, date_dim, customer_address, item
    WHERE
      i_item_id IN (SELECT i_item_id
      FROM item
      WHERE i_category IN ('Music'))
        AND ws_item_sk = i_item_sk
        AND ws_sold_date_sk = d_date_sk
        AND d_year = 1998
        AND d_moy = 9
        AND ws_bill_addr_sk = ca_address_sk
        AND ca_gmt_offset = -5
    GROUP BY i_item_id)
SELECT
  i_item_id,
  sum(total_sales) total_sales
FROM (SELECT *
      FROM ss
      UNION ALL
      SELECT *
      FROM cs
      UNION ALL
      SELECT *
      FROM ws) tmp1
GROUP BY i_item_id
ORDER BY i_item_id, total_sales
LIMIT 100""")
    query5.show()
    query_end_time = time.time()
    query_elapsed_time5 = (query_end_time - query_start_time)
    print("query elapsed time: " + str(query_elapsed_time5))
    print("\n")
    query5.show(1)

    query_list = [query_elapsed_time1, query_elapsed_time2, \
    query_elapsed_time3, query_elapsed_time4, query_elapsed_time5]
    return query_list

def main():
    set_tables()
    query_list = execute_queries()
    print("query execution times")
    print(query_list)

if __name__== "__main__":
  main()
```